ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ОЛЕСЯ ГОНЧАРА

Факультет прикладної математики

Кафедра обчислювальної математики та математичної кібернетики

Дипломна робота

перший (бакалаврський) рівень вищої освіти

спеціальність 113 Прикладна математика

освітня програма Комп’ютерне моделювання та технології програмування

ЗАСТОСУВАННЯ МЕТОДІВ МАШИННОГО НАВЧАННЯ ДЛЯ ПРОГНОЗУВАННЯ ЕЛЕМЕНТІВ КОНСТРУКЦІЙ

Виконавець

студент групи ПА-18-1

Щербак Роман Олексійович \_\_\_\_\_\_\_\_\_\_\_\_\_

Керівник  
Доктор фіз.-мат. наук, професор

А.Є. Шевельова \_\_\_\_\_\_\_\_\_\_\_\_\_

Завідувач кафедри обчислювальної   
математики та математичної кібернетики  
канд. фіз.-мат. наук, доц.

В.А. Турчина \_\_\_\_\_\_\_\_\_\_\_\_\_

Дніпро – 2022

**ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ   
ІМЕНІ ОЛЕСЯ ГОНЧАРА**

Факультет прикладної математики

Кафедра обчислювальної математики та математичної кібернетики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 113 Прикладна математика

освітня програма Комп’ютерне моделювання та обчислювальні методи

ЗАТВЕРДЖУЮ

Завідувач кафедри обчислювальної математики та математичної кібернетики

**\_\_\_\_\_\_\_**\_\_\_\_\_\_Валентина ТУРЧИНА

« » \_\_\_\_\_\_\_\_\_\_\_\_ 2022 р.

***З А В Д А Н Н Я***

***НА ДИПЛОМНУ РОБОТУ***

Щербак Роман Олексійович

1. Тема роботи Застосування методів машинного навчання для прогнозування елементів конструкцій

керівник роботи Шевельова Алла Євгенівна, доктор фіз.-мат. наук, професор

затверджені наказом по Університету від «21» 03 22 року № 303с

2. Термін подання роботи 06.06.2022

3. Вхідні дані до роботи

Експериментальні дані про швидкість наростання втомних тріщин у сплавах, залежно від проміжку значень інтенсивності напружень та циклічних навантажень.

4. Перелік питань, які потрібно розробити

аналітичний огляд літературних джерел з аналізом методів машинного навчання, принципів роботи нейронних мереж, моделей прогнозування швидкості наростання втомних тріщин, застосування нейронних мереж для її розв’язання. Опис моделі розв’язання задачі. Опис програмного забезпечення. Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

не передбачений**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

6. Керівник, консультант з окремих (спеціальних) розділів роботи

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Ініціали прізвище та посада  керівника, консультант | Підпис, дата | |
| завдання видав | завдання прийняв |
|  | Не передбачено |  |  |

1. Дата видачі завдання: 22 березня 2022

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів дипломної роботи | Строк  виконання  етапів роботи | Примітка |
| 1 | Формулювання теми дипломної роботи, складання графіку її виконання | 22.03.2022 | Виконано |
| 2 | Здійснення збору, систематизації та аналізу  наукової інформації про методи та види машинного навчання, нейронні мережі, їх види, алгоритми навчання та застосування для задачі про наростання втомних тріщин | 15.04.2022 | Виконано |
| 3 | Дослідження математичних та комп’ютерних моделей нейронних мереж, математичних моделей задачі про наростання втомних тріщин. Вибір алгоритмів навчання нейронної мережі для поставленої задачі | 29.04.2022 | Виконано |
| 4 | Програмна реалізація нейронної мережі для задачі про наростання втомних тріщин, її навчання за допомогою алгоритмів зворотного поширення помилки та генетичного | 10.05.2022 | Виконано |
| 5 | Проведення відповідних експериментів та чисельних розрахунків з інтерпретацією отриманих результатів | 18.05.2022 | Виконано |
| 6 | Проведення аналізу одержаних результатів та визначення меж їх придатності | 23.05.2022 | Виконано |
| 7 | Оформлення дипломної роботи та супроводжуючої документації | 30.05.2022 | Виконано |
| 8 | Надання в електронному вигляді примірника дипломної роботи до випускової кафедри | 06.06.2022 | Виконано |
| 9 | Надання паперового примірника дипломної роботи з власноручним підписом до випускової кафедри | 15.06.2022 | Виконано |

Студент\_\_\_\_\_\_\_\_\_\_\_\_ Роман ЩЕРБАК

Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_Алла ШЕВЕЛЬОВА

РЕФЕРАТ

Дипломна робота: 77 с., 27 рис., 0 табл., 40 джерел, 1 додаток.

*Об'єкт дослідження:* методи машинного навчання, зокрема, нейронні мережі, форсовані дерева, випадкові ліси, машини опорних векторів, метод *k*-найближчих сусідів як інструменти для прогнозування швидкості наростання втомних тріщин.

*Мета роботи:* дослідження й оцінка міцності та залишкової довговічності елементів конструкцій методами машинного навчання.

*Методи дослідження:* методи машинного навчання, зокрема, нейронні мережі, форсовані дерева, випадкові ліси, машини опорних векторів, метод *k*-найближчих сусідів як інструменти для прогнозування швидкості наростання втомних тріщин.

*Одержані результати та їх новизна:* Було отримано прогнозовану швидкість наростання втомних тріщин в алюмінієвому сплаві 2024-T351, залежно від коефіцієнту інтенсивності напружень та величини циклічних навантажень. Новизна полягає в паралельному застосуванні генетичного алгоритму та алгоритму зворотного поширення помилки для навчання нейронної мережі та порівняння їх результатів і кривих навчання обох алгоритмів.

*Результати досліджень можуть бути застосовані при*  розв’язанні задачі механіки руйнування методами машинного навчання, зокрема, нейронними мережами, прогнозуванні швидкості наростання втомних тріщин за наявності даних про її залежність від інтенсивності навантаження та величини циклічного навантаження.

*Ключові слова:* ДОВГОВІЧНІСТЬ СТРУКТУРНИХ ЕЛЕМЕНТІВ, МАШИНЕ НАВЧАННЯ, НЕЙРОННІ МЕРЕЖІ, ГЕНЕТИЧНІ АЛГОРИТМИ, МЕХАНІКА РУЙНУВАННЯ, НАРОСТАННЯ ВТОМНИХ ТРІЩИН, ІНТЕНСИВНІСТЬ НАВАНТАЖЕННЯ, ЦИКЛІЧНЕ НАВАНТАЖЕННЯ.

ANNOTATION

The graduation research of Roman Shcherbak (Oles Honchar Dnipro National University, Faculty of Applied Mathematics, Department of Calculating Mathematics and Mathematical Cybernetics) deals with crack propagation in metal alloys.

A neural network capable of determining the speed of the crack propagation, depending on the stress intensity factor range and the stress ratio of the loading cycle is developed. The software for creating and learning neural network with the help of a genetic algorithm and backpropagation algorithm is coded with Python programming language, using libraries copy, inspect, math, random, matplotlib and numpy.

The research results can be applied to solving problems of the deformation mechanics field, using machine learning methods and neural networks in particular with the given data consisting of the dependency of crack propagation speed on the stress intensity factor range and the stress ratio of the loading cycle.

The analysis of the precision of the neural network is conducted using the Mean Squared Error (MSE) of the difference between the expected output of the network and the real one.

Рages 77, bibliography 40, pictures 27, tables 0, supplement 1.

Key words: STRUCTURAL ELEMENTS LIFETIME, MACHINE LEARNING, NEURAL NETWORKS, GENETIC ALGORITHMS, DEFORMATION MECHANICS, CRACK PROPAGATION, STRESS INTENSITY, STRESS RATIO.

ЗМІСТ

[РЕФЕРАТ 4](#_Toc105764484)

[ANNOTATION 5](#_Toc105764485)

[ВСТУП 7](#_Toc105764486)

[ПОСТАНОВКА ЗАДАЧІ 10](#_Toc105764487)

[1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ 12](#_Toc105764488)

[1.1 Машинне навчання 12](#_Toc105764489)

[1.2 Нейронні мережі 16](#_Toc105764490)

[1.3 Еволюційні нейронні мережі 35](#_Toc105764491)

[1.4 Генетичні алгоритми для навчання нейронних мереж 37](#_Toc105764492)

[1.5 Прикладна задача про визначення можливості руйнування структурних елементів 40](#_Toc105764493)

[1.6 Застосування нейронних мереж для розв’язання задачі про визначення можливості руйнування структурних елементів 44](#_Toc105764494)

[2 АЛГОРИТМ РОЗВ’ЯЗАННЯ ЗАДАЧІ 49](#_Toc105764495)

[3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 52](#_Toc105764496)

[3.1 Опис функцій програми 52](#_Toc105764497)

[3.2 Інструкція користувача 68](#_Toc105764498)

[4 АНАЛІЗ РЕЗУЛЬТАТІВ РОЗВ’ЯЗАННЯ ЗАДАЧІ 70](#_Toc105764499)

[ВИСНОВКИ 77](#_Toc105764500)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 78](#_Toc105764501)

[Додаток А 84](#_Toc105764502)

# ВСТУП

Для багатьох елементів конструкцій руйнування виникає через втому, поступове накопичення пошкоджень і розпочинається з невеликої тріщини, яка розростається під навантаженням. Втомна тріщина утворюється переважно в концентраторі напружень, тобто в місці пошкодження, яке послаблює область поперечного перерізу матеріалу. Тріщина росте доти, доки матеріал здатний витримувати навантаження. Тому основними факторами, що впливають на міцність елементів конструкцій, є дефекти поверхні деталей, температура та зовнішнє середовище під час експлуатації, характер та умови навантаження.

Найбільші дослідження щодо руйнування структурних елементів конструкцій було проведено в авіаційній галузі, де широко застосовуються сплави алюмінію. Тому найбільш повні експериментальні дані щодо утворення тріщин саме щодо цих матеріалів.

Тому важливо вивчати вплив умов навантаження на ріст втомних тріщин. Найчастіше отримані експериментальні дані містять певний розкид, який слід враховувати при їх аналізі.

Методи оцінки міцності та довговічності опорних елементів конструкцій часто потребують складних розрахунків. Отже, необхідно навчитися розв’язувати задачі механіки руйнування методами машинного навчання, зокрема, нейронними мережами, форсованими деревами, випадковими лісами, машинами опорних векторів та методом *k*-найближчих сусідів, які дозволяють досягти високої точності розв’язків.

Це дозволяє застосовувати зазначені вище методи в різних галузях науки і промисловості, де прогноз довговічності елементів конструкцій має велике значення: металургії, авіабудуванні, залізничному та трубопровідному транспорті, ядерних технологіях тощо. Остаточне рішення прийматимуть експерти, але доступність інформаційних даних підвищить професійний рівень таких рішень.

Метою дослідження є оцінка міцності та залишкової довговічності елементів конструкцій методами машинного навчання. Для досягнення цієї мети необхідно було виконати наступні завдання:

• провести аналітичний огляд методів машинного навчання, а саме навчання з учителем, навчання без учителя, навчання з підкріпленням, нейронних мереж, посилених дерев, випадкових лісів, машин опорних векторів,   
k-найближчих сусідів;

• визначити оптимальну архітектуру еволюційних нейронних мереж;

• спрогнозувати швидкість наростання втомних тріщин (FCG) опорних елементів конструкцій з конкретного матеріалу.

*Об'єктом дослідження* є методи машинного навчання, зокрема, нейронні мережі, форсовані дерева, випадкові ліси, машини опорних векторів, метод *k*-найближчих сусідів як інструменти для прогнозування швидкості наростання втомних тріщин.

*Предметами дослідження* є опорні структурні елементи.

*Актуальність дослідження*: визначення тривалості дії структурних елементів дозволить вчасно визначити необхідність заміни структурного елементу для запобігання його виходу з ладу протягом робочого процесу.

*Результати дослідження можуть бути застосовані*: Методи оцінки міцності та залишкової довговічності відповідальних елементів конструкцій часто вимагають складних і дорогих експериментальних методів для отримання діаграм швидкості наростання втомних тріщин. Тому, маючи експериментальні дані, можна розв’язувати задачі механіки руйнування методами машинного навчання, зокрема, нейронними мережами, форсованими деревами, випадковими лісами, машинами опорних векторів та методом *k*-найближчих сусідів, що дозволяє досягти високої точності. Запропонований підхід до прогнозування швидкості наростання втомних тріщин при регулярному навантаженні методами машинного навчання дозволяє визначити залишкову довговічність на етапі проєктування. Це дозволяє застосовувати зазначені вище методи в різних галузях науки та промисловості, де прогнозування довговічності опорних елементів конструкцій має велике значення.

Дипломна робота складається зі вступу, у якому розкрито питання руйнування структурних елементів, необхідності його дослідження, чотирьох розділів, висновків щодо проведеної роботи, списку використаних джерел та додатків. У першому розділі роботи наведений аналіз предметної області, огляд різних методів машинного навчання, опис принципів роботи та навчання нейронних мереж, огляд моделей наростання втомних тріщин та застосування нейронних мереж для визначення швидкості наростання втомних тріщин. Другий розділ містить опис алгоритму розв’язання цієї задачі, застосований для написання програми. У третьому розділі наведено опис програмного забезпечення, представлено інструкцію користувача. У четвертому розділі розглянуто тестові приклади роботи програми, проаналізовано результати.

# ПОСТАНОВКА ЗАДАЧІ

Задача про наростання втомних тріщин відноситься до задач механіки деформації. Її основна мета – визначення тривалості роботи чи довговічності відповідного структурного елемента (металевого, найчастіше алюмінієвого, сплаву), залежно від навантажень, які здійснюються на нього, характеристик самого елемента (хімічного складу), наявності в ньому попередніх ушкоджень, утворених як під час процесу виготовлення, так і під час процесу експлуатації та швидкості, з якої ці дефекти поширюються (у випадку тріщин, утворених під час навантажень). Розв’язання цієї задачі необхідне для запобігання надмірного зростання втомної тріщини, що може призвести до втрати здатності до виконання своєї роботи структурного елемента, або конструкцій, у якій він знаходиться. У цій роботі реалізовано визначення швидкості наростання втомної тріщини (, де – довжина тріщини, – кількість циклів навантаження) залежно від двох параметрів: проміжку значень коефіцієнту інтенсивності напружень () та величини циклічних навантажень ().

Вхідними даними для задачі наростання втомних тріщин, розглянутої у цій роботі, є вибірка експериментальних даних зі 114 елементів, що відображає залежність швидкості наростання втомної тріщини від проміжку значень коефіцієнту інтенсивності напружень та величини циклічних навантажень.

Для розв’язання цієї задачі було застосовано один із методів машинного навчання – нейронну мережу, створену за допомогою мови програмування Python, що містить два вхідні параметри – проміжок значень коефіцієнту інтенсивності напружень та величину циклічних навантажень – та після проведення необхідних обчислень подає на вихід значення швидкості наростання втомної тріщини.

Задача цієї дипломної роботи – визначити ефективність роботи створеної нейронної мережі на вхідній вибірці даних, поділей на навчальну, що бере безпосередню участь у навчанні, і тестову, що застосовується для перевірки результатів навчання мережі, порівняти між собою два алгоритми навчання, що застосовуються в нейронних мережах, – зворотного поширення помилки та генетичний.

# АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

## Машинне навчання

Машинне навчання – галузь штучного інтелекту, мета якої – розробка методів, що втілюватимуть різноманітні форми навчання, особливо механізми, що здатні утворювати знання, базуючись на прикладах або даних.

Основними методами машинного навчання [1] є кероване навчання (або навчання з учителем). Таким методами надається певна кількість пар *(x, y)*. Задача методів – знайти правильне передбачення *y\**, маючи на вході *x\**. Вхідні значення *x* можуть бути векторами чисел, або ж більшими об’єктами, наприклад документами, зображеннями, ланцюгами ДНК чи графами. Методи керованого навчання використовується для визначення спаму в електронній пошті, розпізнавання облич на зображеннях або в системах визначення медичних діагнозів для пацієнтів.

Інший важливий метод машинного навчання – навчання без учителя. Воно полягає в аналізі даних без чітких позначень, припускаючи, що відомі певні структурні особливості досліджуваних даних. Одним із прикладів є кластеризація, що полягає в поділі інформації на певну кількість груп (кластерів) і знаходженні правила, що дозволить віднести дані, що будуть отримані в майбутньому, до одного з уже утворених кластерів.

Ще один метод – навчання з підкріпленням – полягає в тому, що замість надання чітких прикладів результатів, залежно від заданих вхідних даних, передбачається, що вхідні дані надають лише вказівку, чи є певна дія правильною або ні. У випадку неправильності дії питання про те, як знайти правильну, залишається дійсним. Зазвичай за наявної послідовності входів сигнали про результат дій надаються по завершенню всієї послідовності дій, а не кожної окремої дії [2].

До моделей, що втілюють методи машинного навчання відносяться дерева ухвалення рішень, метод опорних векторів, метод *k*-найближчих сусідів, нейронні мережі тощо.

Дерева ухвалення рішень будуються за допомогою рекурсивної розбивки вибірки даних, яка потім представляється у вигляді логічної структури дерева. Цей процес називається фазою росту. Унаслідок її кожен лист дерева асоціюється з одним атрибутом даних. На фазі обрізки дерева воно узагальнюється шляхом створення піддерева. Цей процес унеможливлює перенавчання. Для обрізки дерева частим орієнтиром є точність класифікації [3].

Класифікація за допомогою дерев ухвалення рішень відбувається сортуванням від кореня дерева до певного листка, що надає класифікацію певної ознаки вхідних даних. Кожен листовий вузол дерева перевіряє значення цієї ознаки, а кожна гілка, що виходить із листового вузла є одним із можливих її значень. Вхідні дані класифікують рухом із кореня дерева, перевіркою значення відповідної ознаки, рухом відповідною гілкою до наступної ознаки та повторенням цього процесу для нового піддерева, утвореного після класифікації за попередньою ознакою. Дерева ухвалення рішень мають широку сферу застосування у практичних задачах. Їх було застосовано для діагностики захворювань, визначення кредитного ризику для заявників на отримання кредиту та багатьох інших задач із реального життя [4].

До методів, що базуються на використанні дерев ухвалення рішень, відносяться підсилені дерева, випадкові ліси та обертові ліси.

Підсилені дерева – дерева ухвалення рішень, для яких застосовано алгоритм градієнтного підсилення. Алгоритм градієнтного підсилення (або алгоритм підсилення) в загальному випадку може використовуватися не лише для дерев ухвалення рішень, а до будь-якого класифікатора, у тому числі нейронних мереж. Через складність утворення достатньо гарного дискримінанту для певної кількості дерев ухвалення рішень, доцільним є утворення декількох простіших, що більш схильні до помилки. Такі дискримінанти називаються слабкими класифікаторами. Підсилення полягає в об’єднанні слабких класифікаторів в один, що буде більш стабільним, точнішим та менш схильним до помилки [5].

Підсилення дерев розпочинається з класифікації певного входу за допомогою звичайного дерева ухвалення рішень. У випадку помилки класифікації входу надається певна вага, шо збільшуються після кожної помилкової класифікації. Зазвичай для класифікації цим методом будуються від 1000 до 2000 дерев.

Після проходження входу кожним із цим дерев, йому надається певне значення оцінки. Якщо класифікація відбувається між двома групами, потрапляння входу до однієї надає події значення 1, до іншої – -1. Нормалізована сума цих оцінок стає фінальним значенням оцінки цього входу. Часто оцінки, що підсумовуються, використовуються у зваженому вигляді. Вища фінальна оцінка позначає належність до групи 1, нижча – до групи 2. Зазвичай вибирається певне значення, що визначає поділ між групами. Події з вищими за це значення оцінками належать до групи 1, з нижчими – до групи 2 [6].

Серед методів підсилення дерев як основні виділяють AdaBoost та стиснення (або *є*-підсилення). Для AdaBoost (Adaptive Boosting, адаптивне підсилення) значення помилки для дерева m визначається як відношення між вагою помилково класифікованих входів та загальною вагою цього дерева. Для обчислення ваг задається параметр  
де попередньо задана стала – значення похибки для дерева m. Вага кожного помилково класифікованого входу множиться на . Ваги нормалізуються таким чином, що сума всіх ваг входів дорівнює 1. Фінальна оцінка обчислюється за формулою .

Для методу стиснення вага кожного помилково класифікованого входу множиться на , де – попередньо задана стала. Фінальна оцінка обчислюється за формулою [7].

Ідея випадкових лісів полягає в побудові певної кількості дерев ухвалення рішень та об’єднання їх у комітет, після чого кожне дерево класифікує об’єкт за певними ознаками. Після цього для об’єкту вибирається клас, за який проголосувала найбільша кількість дерев із комітету [8].

Обертові ліси базуються на чутливості дерев ухвалення рішень до обертання за осями. Тобто, факт того, що класифікація може відрізнятися, залежно від напряму обертання, може бути перевагою в тому випадку, коли дерева використовуються як члени комітету (чи ансамблю), хоча зазвичай це вважається недоліком дерев ухвалення рішень. Дерева прийняття рішень, отримані з оберненої множини вхідних даних, можуть бути точними, при цьому будучи достатньо відмінними [9].

Метод опорних векторів утілює метод машинного навчання з учителем. Основною сферою його застосування є бінарна класифікація. У цьому методі відбувається поділ простору на певну кількість підпросторів, що відповідають класам. Точки, побудовані на поверхні, називаються опорними векторами. Основна робота відбувається саме з цими векторами та з простором, що розташовується в них.

Завдання методу опорних векторів полягає у знаходженні лінійної моделі вигляду , де – вхідний вектор, – параметри, що добираються для певних моделей. Знаходження цих параметрів відповідає розв’язанню задачі опуклої оптимізації. Метод опорних векторів може замінювати нейронні мережі, однак його процес навчання дуже повільний.

Метод *k*-найближчих сусідів [10] є одним із найпростіших алгоритмів класифікації, що часто застосовується також і для задач регресії. Класифікація за цим алгоритмом базується на обчисленні відстані до кожного об’єкта навчальної вибірки та вибору k об’єктів навчальної вибірки, відстань до яких мінімальна.

## Нейронні мережі

Нейронні мережі – це нелінійні математичні засоби, що симулюють обробку інформації людським мозком за допомогою простих об’єктів, що звуться штучними нейронами, установлених у структури, звані шарами. Багатошарові нейронні мережі математично є ациклічними напрямленими графами шарів, кожен вузол таких мереж складається з афінної лінійної функції. Таким чином комбінація всіх шарів утворює нову афінну лінійну функцію. Нелінійність же забезпечують функції активації. Для одношарових нейронних мереж вхід , заданий як вектор ознак ), обробляється нейронами за допомогою ваг , що видає , який урешті-решт приводить до фінального результату з використанням функції активації . Обробка відбувається за допомогою передатної функції, за яку найчастіше беруть оператор зваженої суми, однак за потреби використовують альтернативні варіанти. Процедура в загальному випадку задається таким чином:

або ж *,* де  *–* зміщення (bias) – спроба імітувати людський фільтр, дозволяє припасувати отримані дані до необхідних потреб. Отже, будь-який штучний нейрон можна розглядати як математичну функцію, що приводить до виходу , послідовно застосовуючи лінійну передатну функцію та нелінійну активаційну функцію на входах

Узагальнити цей принцип для багатошарових мереж можна таким чином: входи нейрона стають його виходами після внутрішньої обробки та передаються у вигляді входів наступним нейронами (у мережі прямого зв’язку – нейронам наступного рівня).

Найчастіше тренують нейронні мережі, мінімізуючи функцію помилки (інколи звану функцією вартості чи функцією втрати) . Загалом ця функція обчислює різницю між очікуваними () та отриманими () виходами нейронної мережі. Існують різні види таких функцій, одним із найбільш популярних із них є функція середньоквадратичної похибки . До інших відносять квадратичну помилку , кореневу середньоквадратичну похибку , середню похибку зміщення та абсолютну середню похибку зміщення [24].

Нейронні мережі початково розроблялися як математичні моделі можливостей мозкової обробки інформації. Базова структура нейронної мережі включає невеликі процесори чи вузли, поєднані зв’язками з певною вагою. Вузли відповідають нейронам мозку, тоді як зв’язки відображають силу синапсів між нейронами. Мережа активується за допомогою надання вхідних даних усім або певним вузлам та подальшому поширенні активації через зважені зв’язки між цими вузлами [11].

Поширення активації відбувається за допомогою спрощеної моделі нейрона, розробленої Маккалохом і Піттсом, званої також штучним нейроном, математичним нейроном Маккалоха–Піттса чи формальним нейроном.

За цією моделлю до кожного вузла за допомогою зважених зв’язків надходять вхідні значення від інших вузлів. Процесор утворює лінійну комбінацію вхідних значень із ваговими коефіцієнтами відповідних зважених зв’язків і застосовує до них функцію активації. Значення, отримане за допомогою функції активації, передається як вхідне для наступних вузлів.

Тобто, кожен вузол приймає сигнали від інших вузлів. Вони накопичуються, передаються функції активації, після чого відбувається розрядка у вигляді передачу сигналу іншим вузлам із певною вагою зв’язку. Після цього сигнали можуть розпочати знову накопичуватися у вузлі.

Топологія – спосіб сполучення нейронів, що формують розглянуту нейронну мережу. Топологію можна розглядати як відношення між нейронами мережі, утілене за допомогою зв’язків між ними. Вона має досить важливу роль для функціональності та продуктивності нейронної мережі. Незважаючи на досить поширене використання терміну топологія як синоніму до термінів структура чи архітектура, їхні значення не є чітко визначеними. Тому в деяких роботах терміни структура та архітектура нейронної мережі стосуються не лише топології нейронної мережі, або означають щось цілковито відмінне.

У загальному випадку топологія складається з двох частин: каркасу нейронної мережі та структури зв’язків. Каркас нейронної мережі, у свою чергу, поділяється на кількість шарів у заданій мережі та кількість нейронів на кожному шарі. Структура зв’язків визначає спосіб, за допомогою якого нейрони сполучено між собою. Виділяються чотири види зв’язків між нейронами:

* Міжшаровий (interlayer) – вид зв’язків між нейронами сусідніх шарів.
* Внутрішньошаровий (intralayer) – вид зв’язків між нейронами того самого шару.
* Зв’язок із самим собою (selfconnection) – особливий вид внутрішньошарового зв’язку, зв’язок нейрона зі самим собою.
* Надшаровий (supralayer) – зв’язок між нейронами, що знаходяться на несусідніх і відмінних шарах, тобто зв’язки між ними «перестрибують» один або декілька проміжних шарів.

Кожна нейронна мережа, що поділена на шари, має принаймні один міжшаровий зв’язок. У випадку відсутності достатньої кількості таких зв’язків, над нейронною мережею можливо провести просторове перетворення, що дозволить перетворити інші види зв’язків у міжшарові для новоутвореної мережі, поділеної на шари [12].

Мережі прямого зв’язку – нейронна мережа, у якій рух інформації (у вигляді активації нейронів) відбувається лише в напрямку від вхідного шару до вихідного. У таких мережах наявний лише один тип зв’язків: міжшарові. У випадкові наявності надшарових зв’язків мережа перетворюється на так звану залишкову нейронну мережу, у випадку наявності зв’язків із самим собою чи міжшарових зв’язків, що не ведуть у напрямку від вхідного шару до вихідного – на рекурентну.

За архітектурою нейронні мережі прямого зв’язку поділяються на одношарові та багатошарові. Одношарові містять лише два шари: вхідний і вихідний. Нейрони вхідного шару отримують вхідні сигнали, а вихідного – повертають вихідні сигнали мережі. Синаптичні зв’язки поєднують кожен нейрон вхідного шару з нейроном вихідного, але не у зворотному порядку. Незважаючи на наявність двох шарів (вхідного та вихідного), лише один із них відповідає за обчислення, тому такі нейронні мережі й мають назву одношарових.

Архітектура класу багатошарових мереж містить не лише вхідний і вихідний шари, а також так звані приховані шари, що знаходяться між вхідним і вихідним шаром і здійснюють необхідні проміжні обчислення перед тим як надати вхід на нейрони вихідного шару. Кількість прихованих шарів у багатошарової нейронної мережі не обмежено. Нейрони, що знаходяться на цих шарах, мають назву прихованих вузлів або прихованих нейронів. Ваги зв’язків між вхідним і прихованим шаром нейронів мають назву вхідних-прихованих зв’язків, а між прихованим і вихідним – приховано-вихідних зв’язків. Багатошарові нейронні мережі, що містять велику кількість прихованих шарів часто називають також глибокими нейронними мережами [13].

Як окремий вид нейронних мереж часто виділяють так звані машини екстремального навчання. Це нейронна мережа, що містить один прихований шар, у якій ваги зв’язків, що пов’язують вхідний і прихований шари, обираються випадковим чином, а ваги зв’язків між прихованим і вихідним шаром обчислюються аналітично. Серед переваг машин екстремального навчання виділяють вищу швидкість навчання, порівняно з техніками, що базуються на градієнтному спуску, кращу здатність до узагальнення та високу здатність до запобігання потраплянню в локальні мінімуми. Як функції активації для нейронів, що знаходяться на прихованому шарі можуть застосовуватися сигмоїда, синус, функція Гаусса тощо [14].

Ваги зв’язків між прихованим і вихідним шаром зазвичай отримують за допомогою застосування псевдоберненої матриці Мура-Пенроуза, зважаючи на те, що функції активації нейронів на цьому рівні лінійні [15].



Рисунок 1.1 – Одношарова нейронна мережа



Рисунок 1.2 – Багатошарова нейронна мережа

­

Рисунок 1.3 – Залишкова нейронна мережа



Рисунок 1.4 – Рекурентна нейронна мережа

Нейронні мережі часто навчаються за допомогою алгоритмів, що наближено відповідають градієнтному спуску. Існують різні методи навчання нейронних мереж на великих вибірках за допомогою градієнтного спуску, залежно від принципу, за яким оновлюються ваги цієї мережі. Два основні види називаються дозованим (або партійним) навчанням (batch training) та навчанням онлайн (on-line training). У першому значення параметру, на який змінюється вага нейрона накопичується протягом усього циклу роботи з навчальною вибіркою (так званою епохою) перед тим, як застосовуватися до зміни ваг мережі, тоді як при навчанні онлайн оновлення ваг здійснюється після розгляду кожного навчального прикладу. Проміжну альтернативу часто називають мінідозованим навчанням. У такому випадку значення накопичуються протягом певної кількості прикладів перед тим, як оновити ваги. Параметр часто називають частотою оновлення (або розміром партії). Якщо значення , то мінідозоване навчання перетворюється на навчання онлайн, а якщо , де – розмір навчальної вибірки, – на дозоване. В усіх випадках використовується допоміжний параметр коефіцієнту навчання нейронної мережі, що позначається або . Він дозволяє налаштувати значення зміни ваг, частіше за все, у бік зменшення [16].

Для навчання нейронних мереж зазвичай застосовується алгоритм зворотного поширення помилки. Навчання здійснюється за допомогою ітеративного оновлення ваг зв’язків між вузлами нейронної мережі. Математичний зміст алгоритму полягає в застосуванні негативного градієнту середньоквадратичної функції похибки.

Алгоритм розпочинається з обчислення значення помилки як різниці між очікуваним виходом нейронної мережі та отриманим. Отримане значення поширюється на попередні рівні нейронної мережі, викликаючи коригування ваг зв’язків між нейронами. Алгоритм зворотного поширення помилки застосовується для багатьох задач, однак його швидкість збіжності порівняно невисока, тому розробляються різноманітні модифікації чи інші алгоритми навчання нейронних мереж [17].

Функції активації нейронних мереж поділяються на три основні категорії, що містять певні переваги й недоліки:

1. Бінарна крокова функція – активує нейрон лише в тому випадку, якщо вхідне значення більше чи менше за певний наперед заданий рівень. У такому випадку вузол надсилає те саме значення, що й прийняв. Недолік крокової функції в тому, що вона не підтримує вихід із декількох значень, тобто не підтримує, наприклад, класифікацію в декілька різних категорій.
2. Лінійна функція – функція, сигнал якої пропорційний до вихідного. На відміну від крокової, може приймати значення, відмінні від 0 та 1 (чи -1 та 1, залежно від типу функції). Її недоліки полягають у неможливості застосування зворотного поширення помилки, оскільки похідна функції – стала, у тому, що вихідний шар завжди є лінійною функцією та в неможливості застосування функції для обробки складних даних, оскільки вона є лише лінійною моделлю регресії.
3. Нелінійні функції – дозволяють утворювати складні відображення між входом і виходом нейронної мережі. Їх можна застосовувати також і для складних даних, як-от відео, аудіо тощо. Перевагами є можливість застосування зворотного поширення помилки, оскільки похідні цих функцій – функції, що залежать від вхідних параметрів та використання для навчання зі складним наборів даних із високим рівнем точності [18].

Одним із слабких місць у нейронних є мережах є так звана проблема зникання градієнту. Вона найчастіше виникає в мережах, у яких наявна велика кількість прихованих шарів. Із додаванням більшої кількості прихованих шарів навчання мережі стає все важчим і може призвести до того, що нейронна мережа не зможе завершити своє навчання. Цей випадок і називається проблемою зникання градієнту та є звичним феноменом для глибоких нейронних мереж. Особливо схильними до цієї проблеми є рекурентні нейронні мережі.

Математичний зміст проблеми зникання градієнту полягає в тому, що значення градієнтів нейронної мережі поступово зменшується, що призводить до менших змін у значеннях зв’язків між нейронами, що у свою чергу ускладнює тренування нейронної мережі, зменшуючи швидкість та збільшуючи час її тренування, а часто й зменшуючи точність класифікації. Існує декілька різних способів вирішення цієї проблеми, що дозволяють стабілізувати градієнти та запобігти появі проблеми зникання градієнту. До них відноситься так звана пакетна нормалізація чи комбінація двох алгоритмів навчання та функцій активації [19].

Протилежною до проблеми зникання градієнту є проблема вибухання градієнту. Вона визначається значним збільшенням норми градієнту в ході навчання нейронної мережі. Це стається внаслідок вибуху у значеннях довготермінових компонент, що мають здатність до експоненційного зростання, більшу за короткотермінові. Проблема зникання градієнту, з іншого боку, характеризується експоненційним спаданням значення норми до 0, через що модель не може визначити кореляцію між подіями, що мають певну часову відстань одна між одною [20].

До інших проблем, що виникають під час роботи нейронних мереж відноситься перенавчання. Його сутність полягає в надто близькому підлаштуванні моделі до реальних даних. За наявності перенавчання модель набуває надто заплутаної форми та може не мати жодного відношення до функціональної залежності, яку необхідно визначити. Прямим наслідком цього є нездатність моделі адекватно прогнозувати дані, що не містилися в навчальній вибірці, хоча приклади, задані заздалегідь, відтворюються без помилок. Тобто, відбувається не процес узагальнення відомих прикладів, а просто їх запам’ятовування.

Така проблема виникатиме в математичних моделях будь-якого типу, що налаштовуються через процедуру навчання з учителем, у тому числі й у нейронних мережах. Один зі способів уникнення явища перенавчання – формування множини навчальних прикладів, обсяг якої більший за кількість параметрів моделі, або ж зменшення самої кількості параметрів моделі, з урахуванням відстеження точності моделювання, щоби запобігти недостатній гнучкості мережі через наявність замалої кількості зв’язків у ній [21].

Здатність до узагальнення є ключовою для успішного застосування нейронних мереж на практиці. Для цього вона має гарно наближати дані, що не включено в навчальну вибірку. Щоб мережа мала гарну здатність до узагальнення, необхідно уникати перенавчання, тобто навчати модель визначати лише наявний у навчальній вибірці сигнал, не враховуючи шум. Одним із наслідків цього є необхідність тримати архітектуру нейронної мережі досить простою, оскільки ускладненим моделям значно сильніше властиве перенавчання. Існують різні техніки, що дозволяють покращити здатність мереж до узагальнення, серед яких є різні види перехресного підтвердження (наприклад, рання зупинка), вставлення шуму в дані, упорядкування помилки, ослаблення ваг, оптимізований алгоритм наближення тощо [22].

До основних функцій активації відносяться:

1. Лінійна функція (linear function). Функція задається формулою *a\*x*, похідна цієї функції – де *x* – вхідне значення нейрона. Графіки лінійної функції активації мають такий вигляд:

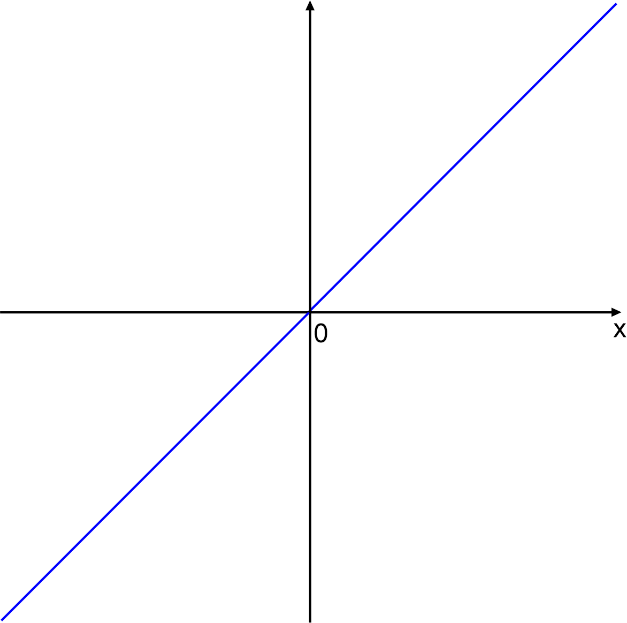


Рисунок 1.5 – Графік лінійної функції активації

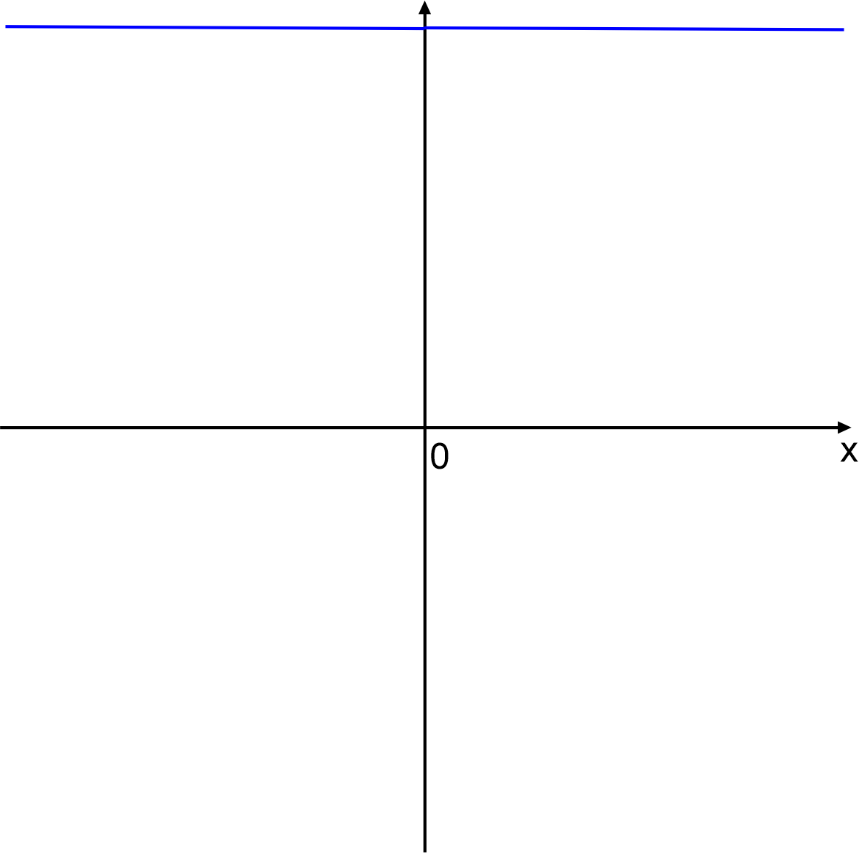


Рисунок 1.6 – Графік похідної лінійної функції активації

1. Зрізаний, або випрямлений лінійний вузол (Rectified Linear Unit, ReLU). Функція задається формулою: , де – вхідне значення нейрона. Похідна задається формулою , якщо та , якщо , де – вхідне значення нейрона. Тобто, ReLU приймає значення 0, якщо на вхід нейрона подається значення x 0 та значення x, якщо на вхід нейрона подається значення x > 0. Переваги цієї функції полягають у вирішенні проблеми зникання градієнту та в достатньо меншій складності обчислення, порівняно з логістичною функцією активації чи гіперболічним тангенсом. До недоліків відноситься те, що ReLU доцільно застосовувати лише на прихованих шарах мережі, те, що нейрони зі вхідними значеннями, меншими чи рівними 0 не будуть змінювати своєї ваги через те, що градієнт ReLU для них буде дорівнювати 0 (так звана проблема «мертвої ReLU») й те, що значення функції не обмежено вгорі, що може викликати вибух активації за достатньо великих значень (проблема зникання градієнту). Графіки зрізаного лінійного вузла мають такий вигляд:



Рисунок 1.7 – Графік функції активації ReLU

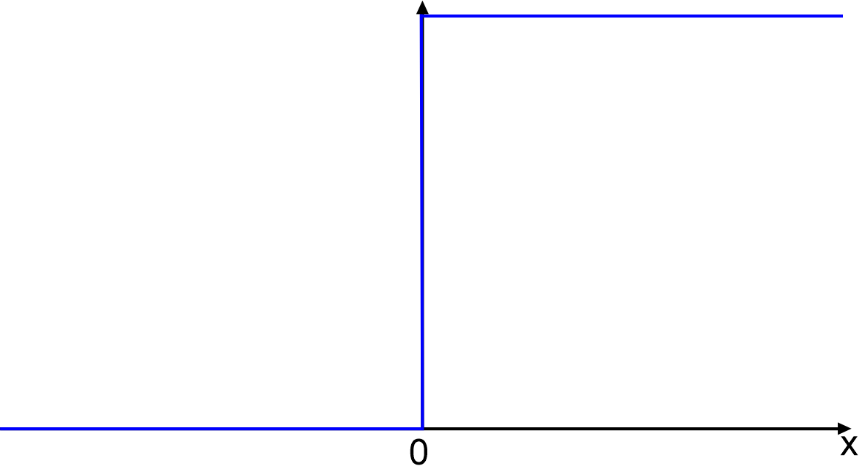


Рисунок 1.8 – Графік похідної функції активації ReLU

1. Нещільний зрізаний лінійний вузол (Нещільна ReLU, Leaky ReLU). Задається формулою , якщо та , якщо , де *x* – вхідне значення нейрона. Похідна має вигляд , якщо та , якщо , де *x* – вхідне значення нейрона. Зазвичай обирають зі значеннями 0.1 або 0.3. Серед переваг виділяють те, що нещільна ReLU дозволяє позбутися проблеми зникання градієнту, а також вирішує проблему «мертвої ReLU». Однак функція не здатна вирішити проблему вибуху градієнту за великих значень . Окрім цього параметр необхідно задавати самостійно, мережа не здатна змінити його у процесі навчання, і те, що похідна нещільної ReLU – лінійна функція. Графіки нещільної ReLU мають вигляд:



Рисунок 1.9 – Графік функції активації Leaky ReLU

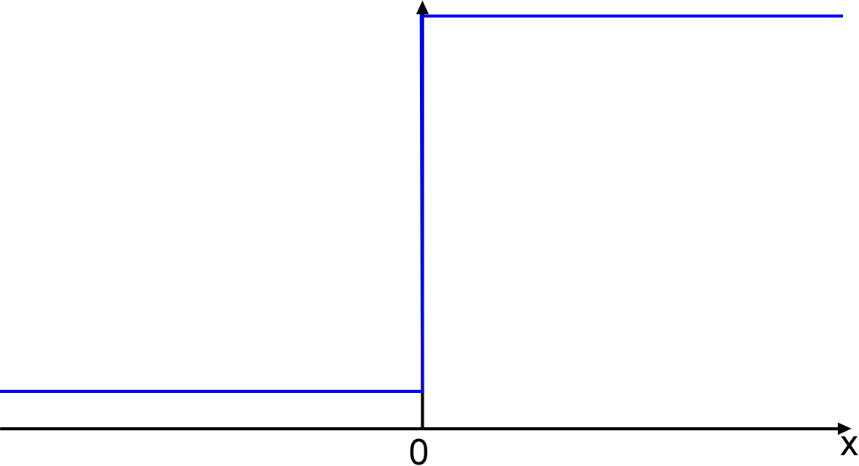


Рисунок 1.10 – Графік похідної функції активації Leaky ReLU

1. Експоненційний лінійний вузол (Exponential Linear Unit, ELU). Функція задається формулою: , якщо та , якщо , де – вхідне значення нейрона. Похідна функції задається як , якщо та , якщо , де – вхідне значення нейрона. До переваг цієї функції відноситься гладкість функції, що викликає поступове збільшення значень функції, порівняно з різким збільшенням зрізаного лінійного вузла та можливість повертати від’ємні значення. Недоліком є необмеженість угорі, притаманна ELU так само, як і ReLU, що може викликати вибух активації. Графіки експоненційного лінійного вузла мають такий вигляд:



Рисунок 1.11 – Графік функції активації ELU

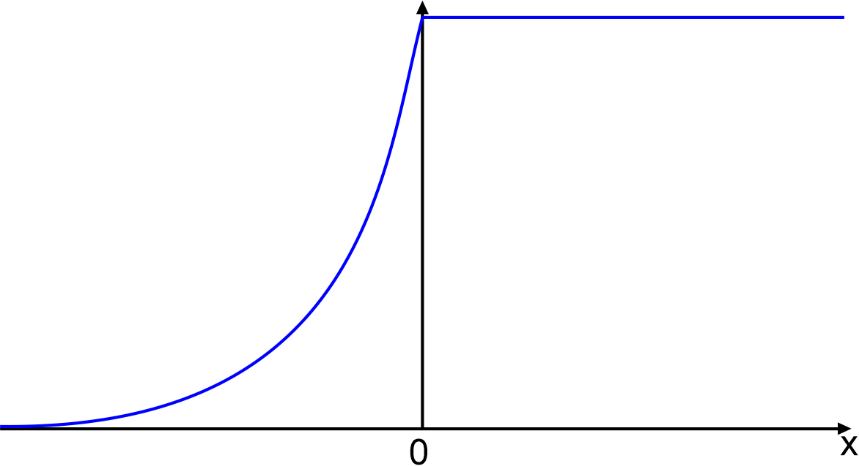


Рисунок 1.12 – Графік похідної функції активації ELU

1. Масштабований експоненційний лінійний вузол (Scaled Exponential Linear Unit). Одна з найновіших активаційних функцій. Задається формулою , якщо та , якщо , де – вхідне значення нейрона. Формула похідної: , якщо та , якщо , де – вхідне значення нейрона. Перевагою є здатність нормалізувати параметри, що дозволяє прискорити збіжність нейронної мережі та не витрачати час на нормалізацію вхідних даних перед поданням їх на вхід нейронної мережі. Інша перевага – неможливість виникнення проблем зникання чи вибухання градієнтів під час застосування SELU. Недоліком є порівняна новизна цієї функції, через що вона вважається не до кінця дослідженою для різних типів архітектур нейронних мереж. Графіки масштабованого експоненційного лінійного вузла мають вигляд:

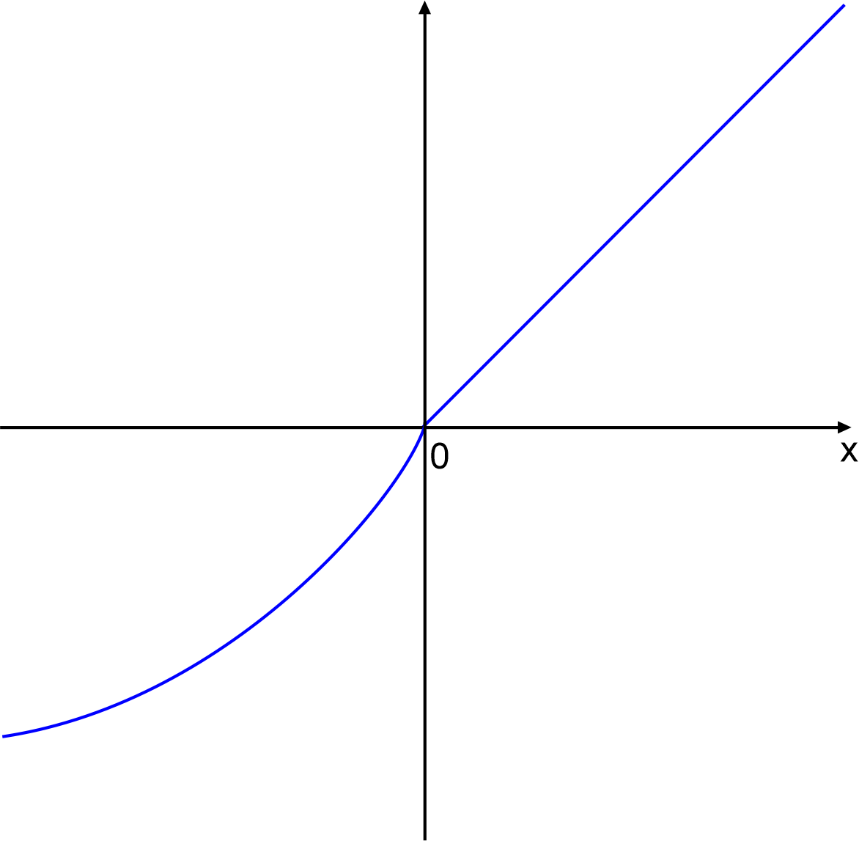


Рисунок 1.13 – Графік функції активації SELU

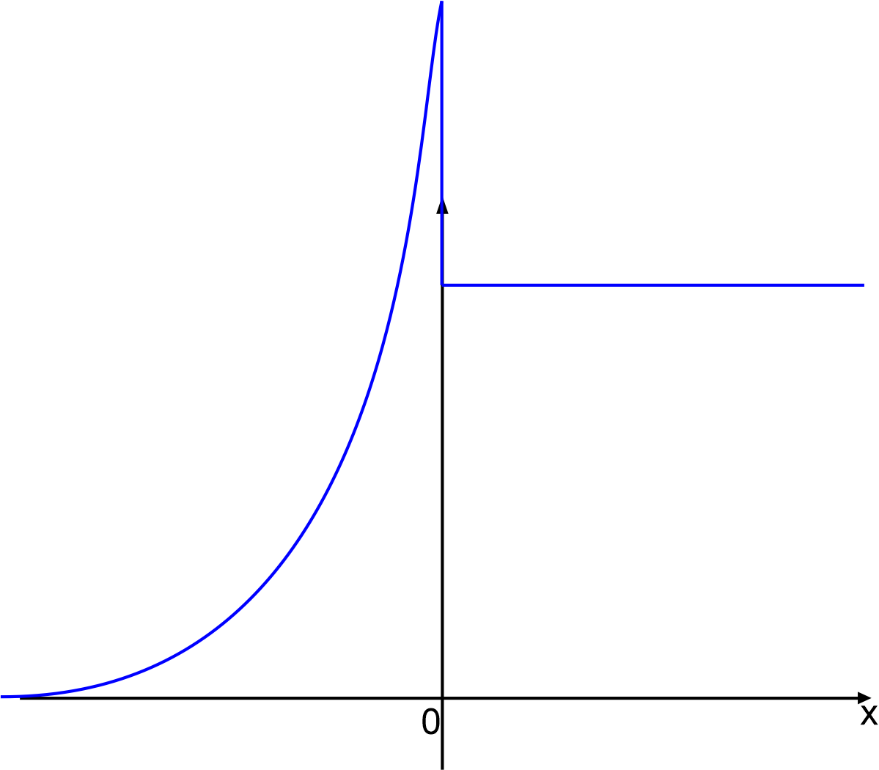


Рисунок 1.14 – Графік похідної функції активації SELU

1. Логістична функція, або сигмоїда (sigmoid). Функція задається формулою , де *x* – вхідне значення нейрона. До переваг цієї функції належать нелінійність, гладкість градієнту, обмеженість значень як угорі, так і внизу, та здатність до класифікації. Серед недоліків виділяються слабка реакція зміни значень функції до зміни значення *x* на обох кінцях, те, що центр значень функції не знаходиться в точці 0, що ускладнює оптимізацію через розкид значень градієнту та схильність до проблеми зникання градієнту. Графіки логістичної функції мають вигляд:



Рисунок 1.15 – Графік логістичної функції активації

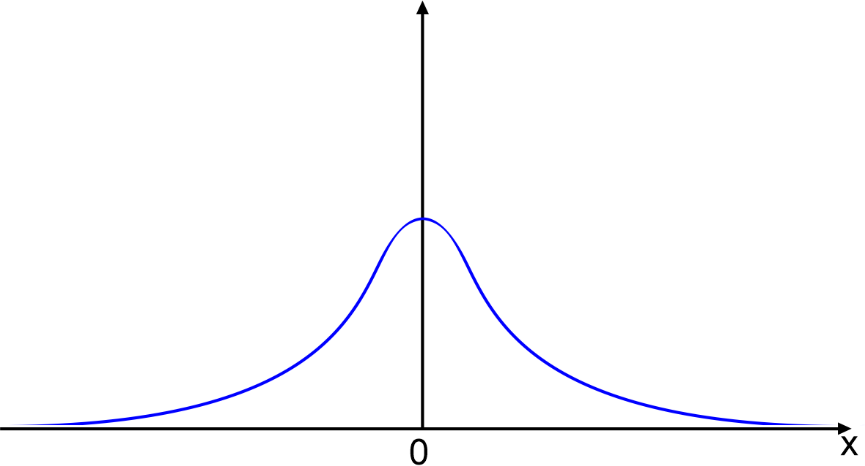


Рисунок 1.16 – Графік похідної логістичної функції активації

1. Гіперболічний тангенс (Tanh). Функція задається формулою , де – вхідне значення нейрона. Похідна функції задається формулою де – вхідне значення нейрона. Перевагою є більша гладкість похідної, порівняно з логістичною функцією та обмеженість значень унизу та вгорі. Недолік – схильність до проблеми зникання градієнту [23]. Графіки гіперболічного тангенсу мають вигляд:



Рисунок 1.17 – Графік функції активації гіперболічний тангенс

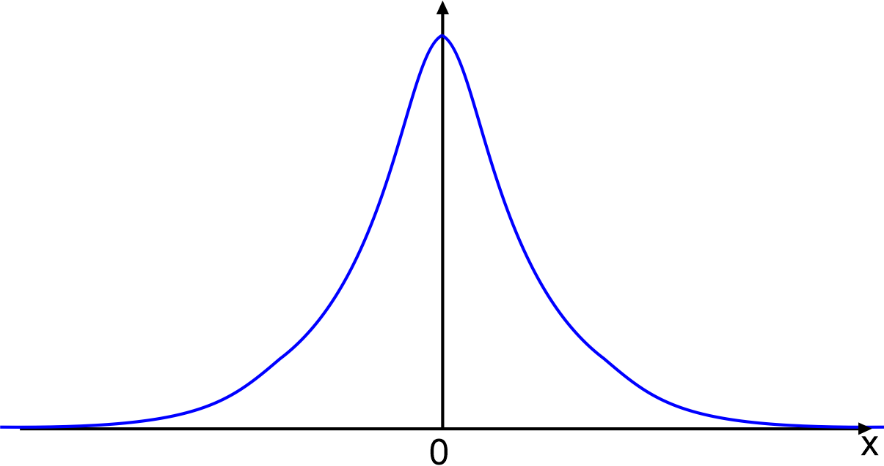


Рисунок 1.18 – Графік похідної функції активації гіперболічний тангенс

1. Softplus. Задається формулою , де – вхідне значення нейрона. Похідна має вигляд , де *x* – вхідне значення нейрона, та збігається з формулою логістичної функції активації. Softplus належить до підкласу активаційних функцій зрізаних вузлів, куди входять також ReLU, ELU, SELU та деякі інші функції. Уперше була визначена як альтернатива для логістичної функції активації чи для гіперболічного тангенсу, однак нині вважається альтернативою для ReLU, що дозволяє боротися з проблемою «мертвої ReLU» [25]. Недолік цієї функції – нелінійність, що призводить до складніших обчислень, у порівнянні з ReLU.

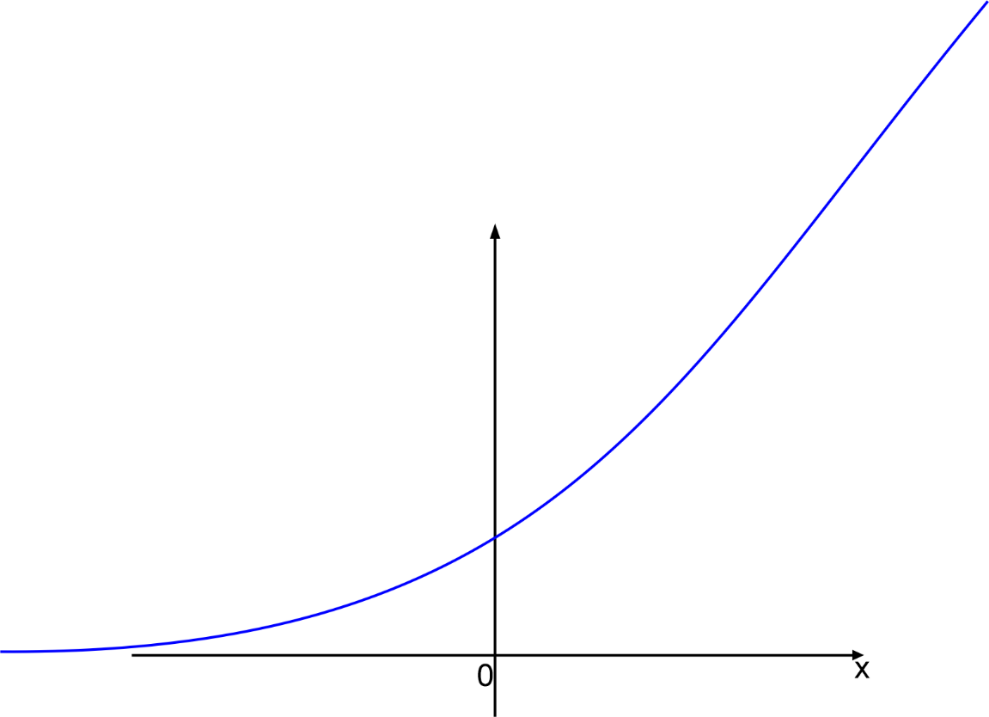


Рисунок 1.19 – Графік функції активації SoftPlus



Рисунок 1.20 – Графік похідної функції активації SoftPlus

## Еволюційні нейронні мережі

Еволюційні алгоритми – стохастичні методи оптимізації, що діють за принципами еволюції у природі: випадково задана популяція особин, що відображає певні точки на просторі пошуку, еволюціонує, слідуючи алгоритму, подібному до дарвінівського виживання найбільш пристосованих. Нові особини утворюються за допомогою генетичних операцій мутації та схрещування. Імовірність виживання певної особини залежить від значення її пристосованості. З високою ймовірністю виживають найбільш пристосовані, найменш пристосовані же швидко зникають. Основні алгоритми, що спираються на цю схему: генетичні алгоритми, еволюційні стратегії та еволюційне   
програмування [26].

Як ще один алгоритм виділяють генетичне програмування. Згадані вище алгоритми вважаються класичними серед еволюційних алгоритмів. До більш нових алгоритмів відносять досить відомі, як-от табу-пошук, алгоритм світлячків та алгоритм мурашиної колонії, так і менш відомі: стрибків жаби, рибної зграї, бджолиної колонії тощо [27].

Ті нейронні мережі, для побудови та навчання яких застосовуються еволюційні алгоритми, називаються еволюційними нейронними мережами. Застосування еволюційних алгоритмів дозволяє досягати кращих результатів у навчанні та узагальненні.

Здійснення еволюції може здійснюватися на трьох різних рівнях, або на їх комбінаціях. Перший рівень – коригування ваг нейронних зв’язків, тобто заміна алгоритму зворотного поширення помилки на один із еволюційних алгоритмів. Другий рівень – еволюція архітектури нейронної мережі. Третій рівень – еволюція правила, за яким навчається нейронна мережа [28].

У цій роботі розглянуто перший рівень еволюції, де нейронна мережа тренується за допомогою кодування ваг зв’язків у хромосому та застосування еволюційного алгоритму для визначення оптимальних значень ваг зв’язків між нейронами. Цей рівень еволюції також можливо покращити, застосовуючи так званий гібридний алгоритм, де еволюція застосовується для знаходження для звуження простору пошуку оптимального розв’язку, після чого оптимальні значення ваг нейронів знаходяться за допомогою алгоритму зворотного поширення помилки.

Другий рівень еволюції полягає в тому, що еволюціонує сама архітектура нейронної мережі. У цьому випадку може змінюватися кількість шарів нейронної мережі, або ж кількість вузлів на кожному шарі. Найбільш ефективно така еволюція працює для невеликих нейронних мереж через порівняну простоту кодування у хромосоми. Для більших нейронних мереж довжина хромосом накладає обмеження, тому часто застосовується інший підхід: пошук найкращого набору правил побудови архітектури нейронної мережі. Саме через те, що розмір правил не збільшується під час збільшення нейронної мережі, цей підхід широко використовується в еволюції архітектури мереж великих розмірів.

На третьому рівні еволюції нейронна мережа еволюціонує за допомогою зміни правила навчання, тобто зміни відбуваються в самому процесі, за яким змінюються ваги зв’язків у нейронній мережі. Зміни можуть полягати в оновленні параметрів алгоритму зворотного поширення помилки, або ж в оптимізації самого алгоритму навчання нейронної мережі.

Наведемо приклад кодування ваг зв’язків нейронної мережі, наведеної на рисунку 1.5 у хромосому, для якої можливо застосувати еволюційний алгоритм. Оскільки значення ваг містять у собі десяткові числа, кодування нейронної мережі за допомогою двійкових чисел є невиправданим. Тому буде використано кодування з використанням дійсних чисел. Відповідна цій мережі хромосома матиме вигляд: *{0.21, 0.4, 0.3, 0.18, 0.75, 0.15, 0.27, 0.55, 0.33, 0.64}.*



Рисунок 1.21 – Приклад нейронної мережі з вагами зв’язків

## Генетичні алгоритми для навчання нейронних мереж

Для навчання еволюційної нейронної мережі було розглянуто та застосовано один із видів еволюційних алгоритмів, що називається генетичний алгоритм.

Генетичний алгоритм – симуляція дарвінівської еволюції. Для нього зберігається популяція хромосом, де кожна хромосома відповідає потенційному розв’язку розглянутої задачі. Для генетичних алгоритмів хромосоми часто називаються ланцюжками. Кожен ланцюжок у свою чергу складається з певної кількості генів, що можуть приймати певну кількість значень, званих алелями. У термінології генетичних алгоритмах гени мають назву ознаки, алелі – значення. Для кожного ланцюжка додається значення придатності, що визначає наскільки пристосованою є особина, кодована цією хромосомою. Придатність визначає відповідна функція придатності. Її можна розглядати як певну міру, яку ми намагаємося максимізувати.

Існує три оператори, що приводять генетичний алгоритм до гарних результатів. Відтворення – процес, за допомогою якого ланцюжки переходять у наступне покоління без змін. Ті особини, що мають найбільше значення придатності мають найбільшу можливість перейти в наступне покоління. Для цього найчастіше застосовується так званий вибір за допомогою рулетки. У ньому колесо рулетки ділиться на певну кількість частин, кожна з яких відповідає кожному окремому ланцюжку. Кожна частина колеса має розмір, відповідний значенню придатності.Розмір частин визначається відповідним значенням придатності. Тому під час обертання колеса найчастіше вибираються найкращі особини. Іншим відомим методом є ранжування, коли ланцюжки відсортовуються за їхнім значенням придатності, і кожному ланцюжку надається кількість нащадків, яких він породжує, відповідно до його місця в ранжуванні. Схрещування або кросовер – процес об’єднання частини одного ланцюжка з частиною іншого. Таким чином ми прагнемо об’єднати гарні частини одного ланцюжка з гарними частинами іншої. Операція схрещування приймає дві особини та утворює дві нові, звані нащадками. Існує багато різних видів схрещування. У деяких із них особини обмінюються між собою певною послідовністю генів, в інших кожен алель добирається окремо. Мутація – процес, за якого випадково обраний ген (або певна їх кількість) у ланцюжку отримує інше значення, відмінне від початкового. Мета цієї операції – ввести новий генетичний матеріал у популяцію чи запобігти його втраті. За допомогою мутації ген може отримати значення, яке не існувало до того в популяції чи те, що було втрачено в результаті відтворення [29].

Основний принцип поєднання нейронної мережі та генетичних алгоритмів полягає в кодуванні інформації про нейронну мережу в геном. Для цього спочатку утворюється певна кількість особин. Продуктивність кожної з утворених таким чином варіантів нейронної мережі визначається після навчання зі зворотним поширенням помилки. Однак, існують певні підходи, що не застосовують алгоритм зворотного поширення помилки, а спираються лише на генетичний алгоритм. Тоді кожна особина оцінюється та впорядковується за рівнем придатності. Визначення придатності може базуватися не лише на продуктивності нейронної мережі, а ще й на розмірі нейронної мережі з метою зробити її якомога меншою. На останньому кроці відбуваються процеси схрещування та мутації, що утворюють особин, які замінюють найгірших, або ж усіх особин у популяції. Проблема об’єднання генетичних алгоритмів та нейронної мережі полягає в кодуванні самої мережі.

Усі підходи до об’єднання нейронних мереж можна розділити на підтримувальні комбінації, де нейронні мережі та генетичні алгоритми діють один за одним та спільні, де вони діють одночасно. За підтримувального підходу генетичні алгоритми та нейронні мережі працюють із різними частинами задачі. Найчастіше генетичний алгоритм застосовують для попередньої обробки набору даних, на якому тренується нейронна мережа. Генетичний алгоритм може бути застосовано для зменшення цього набору даних за допомогою виключення зайвих або непотрібних ознак. За спільного підходу генетичні алгоритми та нейронні мережі використовуються одночасно для утворення популяції нейронних мереж, що найкраще пасує для розглянутої задачі.

Цей підхід може застосовуватися для навчання нейронної мережі за допомогою кодування ваг зв’язків як двійкові або дійсні числа або ж для розвитку та вибору архітектури нейронної мережі, спільно з еволюцією ваг зв’язків, або ж окремо від неї. За комбінованого процесу генетичний алгоритм використовується для тренування нейронної мережі. Серед можливих способів його використання є тренування нейронної мережі за допомогою зміни ваг зв’язків, вибір топології нейронної мережі, тобто встановлення того, яким чином нейрони сполучено між собою та вибір оптимальної конфігурації нейронної мережі й параметрів, що визначають її. Генетичні алгоритми можуть застосовуватися й для оптимізації нейронних мереж, що використовують алгоритм зворотного поширення помилки, замінюючи його як правило навчання, визначаючи найкращу архітектуру нейронної мережі чи адаптуючи саме правило навчання, змінюючи його параметри чи самі формули, що використовуються для оновлення ваг зв’язків [30].

## Прикладна задача про визначення можливості руйнування структурних елементів

Протягом робочого процесу механічні навантаження можуть викликати ушкодження опорних структурних елементів. Причинами таких ушкоджень є невеликі дефекти чи тріщини, наявні з самого початку чи такі, що з’явилися протягом процесу роботи. Зазвичай спочатку тріщини наростають стабільно, тобто з кожним циклом навантаження розмір тріщини незначно збільшується. Це називають наростанням втомної тріщини. Втомна тріщина може зростати протягом тисяч циклів навантаження перед тим, як зростання перетвориться на нестабільне. При досягненні певного критичного ліміту тріщина розпочинає нестабільне зростання, що змушує певні структурні елементи чи всю конструкцію втратити свою придатність до виконання задачі [31].

Тому важливим є вивчення процесу наростання втомних тріщин і розв’язання задачі, що дозволить передбачати його та визначати можливість руйнування опорних структурних елементів, зважаючи на їхній склад і поточний рівень руйнування за умови регулярних циклічних навантажень на них.

Виникнення втомної тріщини зазвичай стається через узаємне зміщення її протилежних берегів. Існує три основні типи таких зміщень: нормальний відрив, поперечний зсув і поздовжній зсув [32].

Англійською мовою ці типи зміщень часто позначають як режими чи способи (modes). Тоді нормальний відрив зветься режимом 1, поперечний зсув – режимом 2, поздовжній зсув – режимом 3.

Нормальний відрив включає в себе всі звичайні навантаження, що зумовлюють появу тріщини [31]. Він виникає тоді, коли береги тріщини зміщуються у своїй площині, а траєкторія руху симетричної сили навантаження знаходиться в межах площини тріщини. Поперечний зсув виникає тоді, коли зсувні навантаження породжують ковзання берегів уздовж напрямку тріщини, тобто в її площині. Поздовжній зсув виникає тоді, коли навантаження діє поза межами будь-якої з площин, примушуючи береги рухатися під прямим кутом до напрямку тріщини. Виділяється також змішаний спосіб, коли декілька з наведених вище зміщень виникають одночасно. Наприклад, одночасне виникнення нормального відриву й поперечного зсуву викликає появу похилої внутрішньої тріщини в матеріалі. У випадку, коли всі три способи виникають одночасно, це називають звичайним або просторовим станом змішаного способу.

Незважаючи на давність дисципліни про визначення наростання втомних тріщин (приблизно 150 років), вона в багатьох аспектах залишається емпіричним мистецтвом, а не наукою. Зародження дисципліни механіки руйнування поставило задачу про передбачення, або ж принаймні розуміння наростання тріщин. Швидко стало зрозуміло, що швидкість наростання не є сталою в часі, а збільшується залежно від інтенсивності навантаження та розміру самої тріщини. Однією з основоположних робіт цієї дисципліни є робота Періса, де було запропоновано використання коефіцієнту інтенсивності напружень Ірвіна (КІН) (у вигляді його проміжку ) для визначення швидкості наростання тріщин . Незважаючи на велику кількість досліджень закону (або моделі) Періса та його похідних, його повне розуміння наразі не досяжне, а також існують інші моделі, що застосовуються для опису швидкості наростання втомних тріщин.

Формула закону Періса має вигляд:

де – порогове значення втомлюваності, – критичний проміжок стійкості матеріалу до розламування (за ), та – так звані сталі Періса. Зазвичай закон Періса найточніший для значень наростання втомної тріщини від до однак на практиці ці обмеження не є точними, і ці значення визначаються з характеристик самого матеріалу [33].

Модель (або закон) Вокера – модифікація закону Періса, що включає додатковий параметр, що включає величину циклічного навантаження , яка відсутня у формулах закону Періса. Для його подання закон Вокера містить параметр

де визначається як . Тоді закон Вокера має вигляд:

або

Модель Вокера дозволяє ввести до розгляду значення , однак потребує введення третього параметру для наближення до кривої залежності до – [34].

Закон Періса є однією з найпростіших та найпоширеніших моделей, однак вона не включає багато факторів, що впливають на наростання втомних тріщин. Наприклад, не має здатності моделювати пороговий регіон наростання втомної тріщини, або ж пришвидшення наростання тріщини в місцях, де КІН наближається до значень критичного проміжку значень КІН – .

Тому існують інші, більш точні й ускладнені моделі. Наприклад, модель Формана:

де та – параметри, визначені матеріалом, – критичний проміжок значень КІН. Якщо ці параметри відомі, модель Формана має здатність моделювати асиметричність навантажень та ефекти прискорення наростання втомних тріщин за проміжку значень КІН, що наближається до критичного [35].

Швидкість зростання тріщини задається за допомогою рівняння:

де *N* – кількість циклів навантаження; *a* – довжина тріщини; , де – найбільше та найменше значення КІН, відповідно; , де – найменше та найбільше навантаження протягом одного циклу.

Зазвичай K подається за допомогою рівняння:

де *Y* – функція виправлення, що залежить від геометрії структури та самої тріщини [36].

## Застосування нейронних мереж для розв’язання задачі про визначення можливості руйнування структурних елементів

Нейронні мережі було неодноразово застосовано для передбачення про руйнування структурних елементів і залишкову тривалість їх роботи. Наведені нижче приклади відображають попередні розробки в цій сфері та результати їх роботи.

У дослідженні 2009 року було розглянуто два алюмінієві сплави, 7020 T7 та 2024 Т3. Дані було отримано за допомогою тестів із декількома різними величинами циклічного навантаження та з навантаженням, що призводить нормального відриву. Для обчислення швидкості зростання тріщини автори дослідження створили нейронну мережу з 9 шарів: 1 вхідного, що містить 3 нейрони, одного вихідного, що містить 1 нейрон та 7 прихованих, що містять, відповідно, 12, 24, 35, 100, 35, 24 та 8 нейронів. Цю структуру було отримано емпірично, з метою надати мережі форми діаманту.

Як вхідні параметри було задано проміжок значень КІН (), максимальне значення КІН () та величину циклічного навантаження (). Вихідним параметром було задано швидкість наростання тріщини (). Вхідні параметри та було нормалізовано до значень із проміжку від 1 до 4, параметр – від 1 до 4, а вихідний параметр – від 0 до 3. Як активаційну функцію було використано гіперболічний тангенс, а похибку було обчислено як суму різниці квадратів між очікуваним та отриманим результатом для всіх даних із вхідної множини. Навчання моделі проводилося для емпіричних даних, у яких величина циклічного навантаження мала значення 0, 0.2, 0.4, 0.6 та 0.8. Задачею мережі було передбачення щодо швидкості наростання втомної тріщини для значення = 0.5. Унаслідок застосування моделі було доведено, що отримані за її допомогою результати добре узгоджуються з експериментальними [37].

В іншій роботі зазначено, що нейронні мережі допомагають розв’язувати задачі такого плану через їхню здатність моделювати дані, не припускаючи, що вони підходять під певний математичний розподіл. У багатьох випадках таке припущення допомагає в моделюванні, однак може призвести до проблем, якщо вибрано неправильний розподіл.

Саму мережу було протестовано на навантаженні, що призводить до нормального відриву, з 3 шарами, на першому з яких міститься 7 вхідних параметрів: температурою, , пороговим значенням , критичним значенням КІН для нормального відриву, значенням для нормального відриву, швидкість наростання тріщини () та кількість циклів (). На прихованому шарі мережі наявно 15 нейронів, а вихідний шар містить 1 нейрон, що відповідає довжині тріщини. Перед тренуванням нейронної мережі всі вхідні та вихідні значення було нормалізовано до проміжку від 0 до 1 з використанням такої формули:

де – нормалізоване значення змінної , що нормалізується, – максимальне значення змінної , – мінімальне значення змінної . Для тестування мережі було застосовано дані для сплаву алюмінію 7075-Т6, що застосовується для авіабудування. Результати було порівняно з експериментальними даними та з передбаченнями з використанням рівнянь, покликаних визначати можливість руйнування структурних елементів. Для порівняння було обрано закон Періса, модель Вокера та модель Формана. У результаті було зроблено висновок, що нейронна мережа мала похибку передбачення, не більшу за 0.043%, а для цього сплаву та вхідних даних найближчою з обраних для порівняння моделей був закон Періса. Також зазначено, що проводилося тестування для інших матеріалів, які не було включено в роботу для скорочення її розміру, однак вказано, що тестування на тих матеріалах зазвичай давало похибку, меншу за 0.5% [38].

У ще одній розглянутій роботі тестування відбувалося на сплаві АІ 2014 за температур у , , , . Параметр для всіх експериментів мав значення 0.1. Особливість цієї роботи полягає в порівнянні результатів обчислення за допомогою нейронної мережі, машини екстремального навчання та моделі пристосовування кривої, що не є методом машинного навчання, однак широко застосовується для передбачення швидкості наростання втомних тріщин. Утворена нейронна мережа містить 4 шарів, на першому з яких наявні 4 вхідні параметри, другий і третій мають 2 нейрони, а останній має 1 вихідний параметр.

Після тестування всіх трьох моделей автори визначили, що методи машинного навчання досить добре пристосовуються до нелінійних залежностей між КІН та швидкістю наростання втомних тріщин, і дають досі непогані результати для різних даних. Порівняння всіх трьох методів привело до висновку, що найточнішою моделлю виявилася модель пристосування кривої, оскільки її середньоквадратична похибка була найменшою. Однак ця модель залежить від початкового припущення про тип функціональної залежності та може помилятися тоді, коли модель неможливо подати у вигляді поліномних функцій, наприклад у випадку логарифмічної залежності. Тому нелінійність у наростанні втомних тріщин краще передбачати за допомогою методів машинного навчання, а не закону Періса чи поліномних моделей пристосування кривої. Як іншу перевагу методів машинного навчання було відзначено їхню здатність ефективно працювати з моделями, що залежать від великої кількості змінних. Серед моделей машинного навчання найкращою виявилася машина екстремального навчання, що мала меншу за нейронну мережу середньоквадратичну похибку та показала себе значно краще для екстраполяції та краще передбачила верхню частину кривої залежності наростання втомних тріщин від КІН, тоді як нейронна мережа показала себе краще на внутрішній частині кривої. Було відзначено також те, що машина екстремального навчання показала себе як найшвидша з усіх трьох застосованих моделей [39].

Як вхідні дані нейронної мережі для експерименту в цій роботі було використано значення, наведені в додатках до джерела [40], де було застосовано декілька видів нейронних мереж для різних видів задач із наростанням втомних тріщин. У першій задачі для визначення довжини тріщини впродовж збільшення кількості циклів було застосовано рекурентну нейронну мережу, що за допомогою зворотних зв’язків на вихідному шарі дозволила вловити вплив довжини тріщини на попередньому циклі навантаження на наступне зростання втомної тріщини. Було показано, що нейронна мережа без рекурентних зв’язків не змогла вловити всі динамічні характеристики наростання тріщин. У другій задачі замість визначення довжини тріщини, нейронна мережа з одним прихованим шаром навчилася визначати сталі Періса та . Для цього з наявних даних про залежність швидкості наростання втомної тріщини в алюмінієвому сплаві 2024-T351 від проміжку значень КІН та величини циклічного навантаження було отримано очікувані значення сталих Періса, після чого утворено дві нейронні мережі, на вхід яким подавалося значення , вихідним для першої було значення , для другої – . Після отримання цих сталих, дані кожного з прикладів вибірки подавалися як вхід для закону Періса, що дозволяло визначити швидкість наростання втомної тріщини та порівняти з результатами з вибірки. Дані цього експерименту наведено в додатках до джерела, тому їх було використано в цій роботі.

В останньому експерименті було утворено нейронну мережу, на якій нейрони вхідного шару мають пам’ять про вхідну інформацію з попередніх циклів, таким чином відображаючи динамічний процес наростання втомних тріщин. Вхідними параметрами мережі є максимальне та мінімальне значення віддаленого навантаження, а також навантаження, що викликає появу тріщини. Як вихідний параметр задано довжину тріщини. Оскільки кожен із них залежить від циклу навантаження, то з’являється необхідність зберігати як теперішні, так і попередні спостереження про процес наростання втомної тріщини. Варто відзначити, що для навчання цієї мережі було застосовано модифікацію алгоритму зворотного поширення помилки під назвою алгоритм зворотного поширення похибки Левенберґа-Маркардта.

1. АЛГОРИТМ РОЗВ’ЯЗАННЯ ЗАДАЧІ

Розв’язання задачі полягає у визначенні залежності швидкості зростання втомної тріщини від величини циклічного навантаження та проміжку коефіцієнту інтенсивності напружень на вибірці зі 114 значень.

Для програми задано було задано параметри мінімальної похибки мережі, при досягненні якої навчання вважається завершеним, error\_threshold: 0.01, кількості епох навчання мережі, при досягненні якої, навчання вважається завершеним, epochs\_threshold: 200, мінімальної початкової ваги нейронів weight\_bottom: -1, максимальної початкової ваги нейронів weight\_upper: 1, коефіцієнту навчання нейронної мережі для алгоритму зворотного поширення помилки eta: 0.005, імовірності схрещування в генетичному алгоритмі crossover\_prob: 0.75, імовірності мутації в генетичному алгоритмі mutation\_prob: 0.25, довжини популяції кожного покоління в генетичному алгоритмі population\_length: 100.

Розпочинається робота з отримання даних із файлу data.txt, де міститься вибірка, використана для навчання мережі. Дані розбиваються на списки set\_data та set\_res, що містять, відповідно, пару значень та відповідне до них значення , .

За довжиною кожного елементу списку set\_data програма знаходить значення змінної in\_parms, що відповідає кількості нейронів на вхідному шарі мережі, а за довжиною кожного елементу списку set\_res – out\_parms, що відповідає кількості нейронів на вихідному шарі мережі.

Списки set\_data та set\_res розбиваються на навчальну та тестову вибірки таким чином, що кожен парний елемент стає частиною навчальної (списки tr\_set\_data та tr\_set\_res), а непарний – тестової (списки test\_set\_data та test\_set\_res).

Далі за допомогою бібліотеки inspect програма отримує доступ до списку всіх функцій модуля activation\_functions, де знаходяться можливі активаційні функції для нейронів мережі та їх похідні, і створює відповідні списки activ\_funcs та activ\_funcs\_ders.

За допомогою функції get\_layer\_num програма просить користувача ввести значення кількості шарів layer\_number та кількість нейронів на кожному з цих шарів, що перетворюється на список neuron\_layer\_quantity.

Наступним кроком утворюється сама нейронна мережа за отриманими від користувача значеннями layer\_number та списку neuron\_layer\_quantity. Отримана нейронна мережа записується у список neurons, який за допомогою модуля copy копіюється у список neurons\_1. Копіювання має на меті запобігти втручання роботи одного алгоритму навчання мережі в роботу іншого.

Після цього визначається змінна u, значення якої відповідає довжині порції навчання в алгоритмі зворотного поширення помилки. Для цього алгоритму її було встановлено як 1, таким чином утілюється онлайн-навчання. За допомогою функції main\_calculation із модуля calculations запускається навчання мережі, обчислюється її вихід на навчальній і тренувальній вибірках, на екран виводяться похибки отриманої мережі, будуються графіки залежності від на обох цих вибірках та крива навчання мережі на навчальній вибірці як залежність середньоквадратичної похибки від номеру циклу навчання.

На наступному кроці значення змінної u вибирається для генетичного алгоритму. У цьому випадку воно дорівнює довжині навчальної вибірки, таким чином утілюється дозоване навчання. Варто зазначити, що генетичний алгоритм знаходить лише локальний оптимум мережі, оскільки значення ваг мереж, які він утворює, завжди знаходяться в межах, заданих у модулі main змінними weight\_bottom та weight\_upper, установлених як -1 та 1, відповідно. Тоді як алгоритм зворотного поширення помилки таких обмежень не має. Так само, як і в алгоритмі зворотного поширення помилки, навчання відбувається за допомогою функції main\_calculation із модуля calculations. По його завершенні обчислюється її вихід на навчальній і тренувальній вибірках, на екран виводяться похибки отриманої мережі, будуються графіки залежності від на обох цих вибірках та крива навчання мережі на навчальній вибірці як залежність середньоквадратичної похибки від номеру циклу навчання.

1. ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
   1. Опис функцій програми

Програма, написана мовою Python, складається з 9 модулів, кожен із яких міститься в окремому файлі та реалізує різні функції, що відповідають за отримання даних із файлу вибірки, створення та навчання нейронної мережі.

1. Модуль main.py

Цей модуль не містить функцій, лише дозволяє задавати параметри, необхідні для роботи нейронної мережі, серед яких eta (коефіцієнт навчання нейронної мережі), error\_threshold (похибка, яку не має перевищувати результат роботи нейронної мережі), epochs\_threshold (максимальна кількість епох навчання нейронної мережі), u (кількість елементів у кожній партії, що використовується для зміни ваг нейронної мережі протягом однієї епохи обробки вхідних даних), weight\_bottom та weight\_upper (межі, у яких ініціалізуються ваги зв’язків між нейронами, а також їхні зміщення), crossover\_prob, mutation\_prob, population\_length, ga\_weight\_bottom та ga\_weight\_upper (необхідні в генетичному алгоритмі як імовірність схрещування, імовірність мутації, довжина популяції та межі, у яких ініціалізуються ваги зв’язків між нейронами, а також їхні зміщення в усіх мережах із популяції, відповідно) та накопичувати деякі змінні, необхідні для її роботи: neurons\_created, що відповідає за кількість утворених нейронів та neurons, що є списком, у якому зберігаються утворені нейрони. У цьому модулі запускаються функції get\_dataset (з модулю get\_data), що дозволяє отримати дані, які буде передано нейронній мережі, get\_layer\_num (з модулю create\_network), create\_neurons (з модулю create\_network), що дозволяє створити необхідну кількість нейронів, відповідно до заданих вище параметрів, main\_calculation(з модулю calculations), що відповідає за процес навчання мережі, будує графіки отриманих унаслідок цього навчання результатів. Принцип роботи модуля полягає в утворенні початкової нейронної мережі, відповідно до заданих значень, поділі вхідної вибірки на навчальну й тестову, передачі мережі на навчання на базі навчальної вибірки, виведенням отриманих результатів, загальної помилки нейронної мережі та кількості епох навчання, перевірки навчання на тестовій вибірці, відкладення на графіку кривої навчання мережі та порівняння отриманих результатів обчислення мережі з тренувальною та навчальною вибірками.

1. Модуль get\_data.py

Модуль дозволяє отримати дані вибірки, нормалізувати їх, визначити й додати до списку всі активаційні функції, задані у модулі activation\_functions. Містить функції:

* + get\_dataset (file\_name) – отримує вхідні та очікувані вихідні дані вибірки з файлу file\_name.txt, де file\_name – вхідний параметр функції, записати їх у списки й повернути для подальшої роботи;
  + normalize (set\_list) – нормалізує вхідний список set\_list за формулою максимінної нормалізації з використанням функцій бібліотеки numpy, що перетворює вхідний список на матрицю, транспонує її, визначає максимальне та мінімальні значення кожного з її рядків, застосовує до кожного з елементів відповідну формулу, після чого транспонує матрицю у зворотному напрямку, перетворює її на список і повертає;
  + get\_activation\_funcs() – утворює та повертає списки activ\_funcs, що містить усі функції активації, задані в модулі activation\_functions, та activ\_funcs\_ders, що містить похідні відповідних функцій.

1. Модуль neuron\_class.py

Цей модуль містить клас Neuron, що моделює процес роботи звичайного нейрона, з урахуванням необхідної для нейронної мережі специфіки. Цей клас містить такі параметри: index (порядковий номер нейрона), layer (номер рівня мережі, на якому знаходиться цей нейрон), bias (значення зміщення нейрона), error (значення похибки нейрона, що застосовується для алгоритму зворотного поширення помилки), S (локальне індуковане поле нейрона), activ\_func (функція активації нейрона). У цьому класі реалізовано такі функції:

* \_\_init\_\_ (self, num\_ind, num\_layer, func, func\_der, \*args) – конструктор класу, що дозволяє утворювати новий нейрон та передавати його полю index отримане значення num\_ind, а його полю layer отримане значення num\_layer. Також ініціалізує поле bias випадковим значенням із проміжку [args[0];args[1]] (якщо передано значення args та нейрон не знаходиться на 0 рівні. У випадку відсутності значення bias, очікується, що його буде задано пізніше з використанням функції set\_bias), поле activ\_func значенням вхідного параметру func, поле activ\_func\_der значенням вхідного параметру func\_der та утворює додаткові параметри weights (список ваг нейрона), exit\_values (список вихідних значень нейрона);
* \_\_repr\_\_(self) – допоміжна функція, що дозволяє визначити, яким чином виводити об’єкти класу на екран. Реалізовано у вигляді “{номер індексу}\_{номер рівня}”;
* get\_index(self) – повертає значення індексу нейрона;
* get\_layer(self) – повертає значення рівня нейрона;
* add\_random\_weights(self, next\_layer\_quant, \*args) – додає ваги зв’язків із нейронами наступного рівня (їхню кількість задає параметр next\_layer\_quant) випадковим чином. Якщо в списку args передано значення, то випадкова ініціалізація ваг відбувається в проміжку цих значень, у протилежному випадку – проміжок задається як [-1;1];
* add\_weights\_from\_arr(self, weight\_arr) – додає ваги зв’язків із нейронами наступного рівня зі списку weight\_arr, необхідна для роботи генетичного алгоритму;
* get\_weight(self, ind) – повертає вагу зв’язку між цим нейроном та конкретним нейроном наступного рівня, визначеним параметром ind;
* get\_weights(self) – повертає весь список ваг зв’язків цього нейрона та нейронів наступного рівня;
* update\_weight(self, ind, value) – оновлює вагу зв’язку між цим нейроном і конкретним нейроном наступного рівня, визначеним параметром ind, додаючи до цієї ваги параметр value;
* get\_bias(self) – повертає значення зміщення цього нейрона;
* update\_bias(self, value) – оновлює зміщення нейрона, додаючи до нього параметр value;
* set\_bias(self, value) – призначає зміщенню нейрона значення value;
* set\_error(self, value) – оновлює значення похибки нейрона, додавши до нього параметр value;
* get\_error(self) – повертає значення похибки нейрона;
* set\_exit\_value(self, ind, value) – накопичує у списку значення, передане нейрону наступного рівня, визначеному параметром ind. Якщо такого значення у списку немає, додає його, призначаючи йому значення value, у протилежному випадку – додає до наявного значення value;
* get\_exit\_value(self, ind) – повертає значення передане нейрону наступного рівня, визначеному параметром ind;
* update\_S(self, value) – оновлює значення локального поля нейрона, додаючи до нього значення value;
* set\_S(self, value) – призначає значення локального поля нейрону значення value. Застосовується для скидання цих значень після проходження одного циклу навчання;
* get\_S(self) – повертає значення локального поля нейрона;
* set\_activation\_function(self, func) – визначає активаційну функцію нейрона;
* set\_activation\_function\_der(self, func) – визначає похідну активаційної функції нейрона;
* get\_activation\_function(self) – повертає активаційну функцію нейрона;
* get\_activation\_function\_der(self) – повертає похідну активаційної функції нейрона;
* get\_activation\_function\_value(self, x) – повертає значення функції активації нейрона в точці x;
* get\_activation\_function\_der\_value(self, x) – повертає значення похідної функції активації нейрона в точці x;
* to\_zero(self) – скидає значення error, S та exit\_values після обробки однієї партії з вибірки даних.

1. Модуль create\_network.py

У цьому модулі створюється початкова нейронна мережа, яку й навчають за допомогою алгоритмів зворотного поширення помилки та генетичного алгоритму. Модуль містить такі функції:

* get\_layer\_num(parms\_in, parms\_out) – запитує в користувача бажану кількість проміжних рівнів нейронної мережі, після чого для кожного з них запитує бажану кількість рівнів та додає до списку на відповідне до номеру цього шару місце, ураховуючи, що на 0 місці знаходиться значення parms\_in (кількість вхідних параметрів мережі та, відповідно, кількість нейронів на 1 шарі), а на останньому – parms\_out (кількість вихідних параметрів мережі та, відповідно, кількість нейронів на останньому шарі). Повертає введене значення кількості проміжних шарів та утворений список із розподілом нейронів за ними;
* create\_layer(num, quant, layer, func, func\_der) – утворює новий шар нейронів із номером у мережі, заданим параметром layer, активаційними функціями та їх похідними, заданими, відповідно, параметрами func і func\_der, довжина шару задається параметром num у вигляді списку, що повертається на вихід із функції. quant задає кількість нейронів мережі, утворених на попередніх шарах;
* add\_weights(neur\_arr, neur\_layer\_arr, \*args) – додає ваги до нейронів, ітеруючи заданий двовимірний список нейронів neur\_arr, neur\_layer\_arr задає кількість нейронів на кожному шарі, args – необов’язковий параметр, що може містити список, з якого додаються ваги, якщо програма не отримує його, додавання ваг відбувається випадковим чином, у протилежному випадку вони отримуються зі списку, що міститься в args[0] (якщо довжина args – 1), або ж із якого проміжку будуть випадково задаватися ваги зв’язків між нейронами (якщо довжина args – 2);
* create\_neurons(parms\_in, parms\_out, layer\_num, neur\_layer\_arr, func, func\_der, \*args) – утворює необхідну кількість нейронів (об’єктів класу Neuron) на кожному шарі, послідовно викликаючи функцію create\_layer, та записує їх у список, після чого додає ваги всім утвореним нейронам і повертає утворений список. Кількість нейронів на першому шарі задається параметром parms\_in, на останньому parms\_out, кількість проміжних шарів мережі визначає параметр layer\_num, розподіл нейронів за цими шарами – neur\_layer\_arr, активаційні функції для кожного з типів шарів обираються разом із їхніми похідними з параметрів func та func\_der. Параметр args – необо’язковий, визначає, чи будуть додаватися ваги у функції add\_weights випадковим чином, або ж із списку args[0] (якщо довжина args – 1), або ж із якого проміжку будуть випадково задаватися ваги зв’язків між нейронами (якщо довжина args – 2).

1. Модуль calculations.py

Цей модуль проводить необхідні розрахунки результатів роботи нейронної мережі з заданими вагами, визначає результат її роботи та відхилення від очікуваного результату. Містить такі функції:

* calculate\_layer(curr\_layer, next\_layer, in\_arr, learning\_algorithm) – обчислює та повертає у вигляді списку результат роботи одного шару мережі, якщо значення змінної learning\_algorithm, що визначає алгоритм, за яким навчається нейронна мережа, відповідає зворотному поширенню помилки, то зберігає додаткові значення, необхідні для навчання за допомогою зворотного поширення помилки значення (локального індукованого поля, суми похідних функцій активації кожного нейрона в точці, що відповідає його локальному індукованому полю). На вхід подаються списки curr\_layer та next\_layer, що відповідають шару мережі, що розглядається зараз та наступному за ним, відповідно, in\_arr, що відповідає вектору вхідних значень від curr\_layer до next\_layer та learning\_algorithm;
* calculate\_result(neur\_arr, in\_arr, learning\_algorithm) – ітеруючи списком нейронів neur\_arr, послідовно викликає функцію calculate\_layer, робота якої визначається значенням змінної алгоритму навчання мережі learning\_algorithm, та обчислює результат роботи мережі. in\_arr – вхідний вектор, що змінюється після кожного запуску функції calculate\_layer, останній із них повертається. Після кожної ітерації значення індукованого локального поля кожного нейрона отримує значення 0;
* calculate\_error(exp\_res, res, learning\_algorithm) – обчислює та повертає суму різниць між відповідними компонентами вектору очікуваного результату exp\_res та отриманого внаслідок роботи мережі вектору res, залежно від значення learning\_algorithm може повертати список, що містить різницю між усіма значеннями очікуваного та отриманого результату роботи нейронної мережі, що необхідно для навчання зворотним поширенням помилки;
* calculating\_cycle(set\_data, set\_res, neur\_arr, learning\_algorihm) – послідовно запускає функції calculate\_result та calculate\_error, повертає результати їхньої роботи. На вхід приймає вхідний вектор set\_data, вектор очікуваного результату set\_res, список нейронів мережі neur\_arr та змінну, що визначає алгоритм навчання learning\_algorithm;
* learning\_cycle(neur\_arr, neur\_layer\_arr, set\_data, set\_res, eta, batch, learning\_algorithm, \*args) – виконує одну епоху навчання нейронної мережі за допомогою функції calculating\_cycle. Залежно від параметру learning\_algorithm декілька разів протягом епохи (за досягнення значення довжини партії batch) запускає навчання за допомогою зворотного поширення помилки, застосовуючи функцію backpropagation з модуля learning\_algorithms або ж по завершенню епохи за допомогою функції genetic з модуля learning\_algorithms. Інші вхідні параметри: neur\_arr – список, що містить усі нейрони мережі, neur\_layer\_arr – список, що містить розподіл кількості нейронів за шарами мережі, set\_data – вхідні дані нейронної мережі, set\_res – очікуваний результат роботи мережі, eta – коефіцієнт навчання мережі, args – необов’язкові параметри мережі – для алгоритму зворотного поширення помилки містять два списки: той, що містить усі значення , на які потрібно оновити зв’язки між нейронами і той, що містить усі значення , на які потрібно оновити зміщення нейронів, для генетичного алгоритму – п’ять чисел, що відповідають верхній і нижній межі значень ваг зв’язків між нейронами та зміщення цих нейронів, імовірності мутації, імовірності схрещування та розміру популяції, та два списки, що містять активаційні функції та їхні похідні. Якщо значення learning\_algorithm відповідає зворотному поширенню помилки, функція повертає результат роботи мережі, її помилку та оновлені протягом роботи списки з параметру args, якщо – генетичному алгоритму, – оновлену нейронну мережу у вигляді списку нейронів, результат роботи мережі та її похибку;
* learning\_process(error\_threshold, epochs\_threshold, set\_data, set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm, \*args) – відповідає за навчання нейронної мережі за заданим алгоритмом, визначеним змінною learning\_algorithm. За допомогою циклу функція проводить необхідну кількість епох навчання, переглядаючи вхідну вибірку за елементами, обчислюючи результат і похибку на даному прикладі з вибірки функцією calculate\_cycle (з модуля calculations.py), та кожного разу, коли досягається значення довжини партії, запускає алгоритм зворотного поширення помилки, відповідно до значення параметру learning\_algorithm, використовуючи функцію backpropagation (з модуля learning\_algorithms.py). Після неї нейронна мережа має змінені ваги, на яких продовжуються обчислення до наступного досягнення значення batch. Генетичний алгоритм може запускатися за допомогою функції genetic (з модуля learning\_algorithms.py) після завершення кожної епохи. Після нього утворюється нова нейронна мережа, яка проводить обчислення в наступній епосі. Завершення роботи відбувається, якщо похибка мережі стає меншою за задане значення error\_threshold або кількість епох навчання стає більшою за задане значення epoch\_threshold. Інші вхідні параметри: set\_data – вхідні дані нейронної мережі, set\_res – очікуваний результат, який необхідно отримати, neur\_arr – список нейронів мережі, neur\_layer\_arr – список, що визначає розподіл нейронів за шарами, eta – коефіцієнт навчання, args – необов’язковий параметр мережі, необхідний для навчанням генетичним алгоритмом, – п’ять чисел, що відповідають верхній і нижній межі значень ваг зв’язків між нейронами та зміщення цих нейронів, імовірності мутації, імовірності схрещування та розміру популяції, та два списки, що містять активаційні функції та їхні похідні;
* calculate\_test\_set(neur\_arr, set\_data, set\_res) – визначає якість навчання нейронної мережі, записаної в список neur\_arr, за допомогою проведення циклу обчислення без оновлення ваг (за допомогою функції calculating\_cycle), визначення його результату та обчислення відповідної похибки нейронної мережі. Інші вхідні параметри: set\_data – вхідні дані нейронної мережі, set\_res – очікуваний результат, який необхідно отримати;
* main\_calculation(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, test\_set\_data, test\_set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm, \*args) – відповідає за навчання нейронної мережі, заданої списком нейронів neur\_arr, з застосуванням алгоритму, відповідно до значення параметру learning\_algorithm (за допомогою функції learning\_process), виведення похибки отриманого результату на навчальній вибірці tr\_set\_data, порівняно з очікуваним у ній результатом tr\_set\_res, побудові графіка відповідності між очікуваним та отриманим результатом, кривої навчання нейронної мережі, обчислення та виведення похибки отриманого з мережі результату на тестовій вибірці test\_set\_data, порівняно з очікуваним у ній результатом test\_set\_res, побудові графіка відповідності між очікуваним та отриманим результатом. Інші вхідні параметри – error\_threshold – задане значення похибки нейронної мережі, при досягненні якого навчання завершується, epochs\_threshold – задане значення кількості епох навчання нейронної мережі, при досягненні якого воно завершується, neur\_layer\_arr – список, у якому записано кількість нейронів на кожному шарі мережі, eta – коефіцієнт навчання нейронної мережі, batch – довжина порції навчання, після досягнення якої відбувається оновлення ваг зв’язків між нейронами, args – необов’язковий параметр мережі, необхідний для навчанням генетичним алгоритмом, – п’ять чисел, що відповідають верхній і нижній межі значень ваг зв’язків між нейронами та зміщення цих нейронів, імовірності мутації, імовірності схрещування та розміру популяції, та два списки, що містять активаційні функції та їхні похідні.

1. Модуль genetic\_algorithm\_functions.py

Цей модуль необхідній для навчання нейронної мережі за допомого генетичного алгоритму. Містить такі функції:

* make\_chromosome(neur\_arr) – з кожного нейрона заданого списку neur\_arr знаходить усі ваги зв’язків і записує їх у кортеж, що відповідає хромосомі, та повертається після того, як ітерування списком завершується;
* create\_initial\_population(initial\_population\_length, neur\_arr, neur\_layer\_arr, set\_length, set\_data, set\_res, w\_bottom, w\_upper, func\_arr, func\_der\_arr) – утворює початкову популяцію особин кількістю initial\_population\_length, одна з яких відповідає початковій випадковим чином заданій нейронній мережі, записаній у список neur\_arr, усі інші утворюються за допомогою функції create\_neurons (модуль create\_network.py) з активаційними функціями та їх похідними, заданими, відповідно списками func\_arr та func\_der\_arr. Для кожної нейронної мережі обчислюється значення пристосованості (у вигляді похибки), після чого кожна мережа записується за допомогою функції make\_chromosome записується в кортеж, що моделює хромосому. Цей кортеж разом зі значенням пристосованості записуються у словник, який повертається з функції. До інших вхідних параметрів належать neur\_layer\_arr – список, що визначає розподіл нейронів за шарами, set\_length – довжина вхідної вибірки, set\_data – вхідні дані нейронної мережі, set\_res – очікуваний вихідний результат нейронної мережі, w\_bottom – нижня межа можливих значень ваг зв’язків між нейронами, w\_upper – нижня межа можливих значень ваг зв’язків між нейронами;
* norm\_fitness(population) – функція створює та повертає новий словник, у якому внормовано значення придатності кожної особини популяції population за формулою  
  де – імовірність вибору і-го елемента словника, – експонента від значення і-го елемента словника, – сума експонент значень усіх елементів словника. Формула застосовується з метою використати принцип рулетки, за якого особина, пристосованість якої найбільше відповідає умовам задачі (у цьому випадку мінімізації цільової функції), має найбільшу ймовірність бути вибраною, тоді як та, пристосованість якої найгірша має найменшу, але ненульову ймовірність;
* parent\_selection(normalized\_population) – функція вибирає з унормованої популяції normalized\_population двох осіб, які стануть батьками майбутніх нащадків, за принципом рулетки та записує їх у список, який повертається з функції;
* crossover(chromosome\_1, chromosome\_2) – у функції відбувається схрещування двох особин chromosome\_1 та chromosome\_2 за таким алгоритмом: вибирається найбільша можлива довжина ділянок схрещування за формулою

,

де – довжина хромосоми 1. Та кількість цих ділянок за формулою

.

Після цього кожна хромосома розбивається на відповідну кількість ділянок та вибирається відповідна кількість випадкових індексів, що визначають, якими саме ділянками хромосом відбудеться обмін. Ділянки з однаковими індексами переходять від однієї особи до іншої та навпаки. Після схрещування хромосоми знову об’єднуються в єдиний кортеж і повертаються з функції;

* mutation(chromosome, w\_bottom, w\_upper) – відбувається мутація хромосоми особини-дитини chromosome за принципом випадкового скидання (random resetting). Для його втілення спочатку хромосома перетворюється з кортежу на список, визначається кількість значень, які буде скинуто за формулою

де – довжина хромосоми. Усі індекси списку, що містить хромосому записуються в окремий список, з якого вибирається кількість елементів . Після цього елементам списку, що містить хромосому, призначаються нові випадкові значення у проміжку [w\_bottom;w\_upper]. Після оновлення всіх обраних значень хромосома знову перетворюється на кортеж та повертається з функції;

* offsprings\_creation(normalized\_fitness\_population, w\_bottom, w\_upper, crossover\_prob, mutation\_prob) – об’єднує та послідовно запускає функції parents\_selection, crossover та mutation за таким алгоритмом: з унормованого словника normalized\_fitness\_population, що містить кожну хромосому, поєднану з її значенням придатності, випадковим чином обираються дві особини-батьки за допомогою функції parent\_selection. Обирається випадкове значення схрещування, значення якого знаходяться в діапазоні від 0 до 1. Якщо воно менше за попередньо задане значення crossover\_prob, то відбувається схрещування особин-батьків за допомогою функції crossover. Для кожної з утворених унаслідок цього особин обирається своє значення мутації, значення якого знаходяться в діапазоні від 0 до 1. Якщо воно менше за попередньо задане значення mutation\_prob, для кожної з особин незалежно відбувається процес мутації за допомогою функції mutation. Функція повертає обидві нові особини, що утворилися;
* create\_new\_network(chromosome, neur\_arr) – відбувається утворення нової нейронної мережі за допомогою ітерування «еталонним списком» нейронів neur\_arr та утворенням нового нейрона (як об’єкту класу Neuron) та додаванням до нього ваг із хромосоми chromosome за допомогою функції класу add\_weights\_from\_arr. Кожен шар мережі записується у список, який у свою чергу записується у список, що відповідає новій мережі, яка й повертається з функції;
* choose\_best\_network(population, neur\_arr, set\_data, set\_res) – відбувається вибір найкращої мережі з популяції. Для цього за допомогою функції create\_new\_network з «еталонного списку» у neur\_arr та особини з популяції population утворюється нова нейронна мережа, за допомогою функції calculating\_cycle (модуль calculations.py) відбуваються обчислення цією мережею на вибірці зі вхідних даних set\_data, визначається їхня похибка, відповідно до значень вектору бажаного виходу set\_res. По завершенню ітераційного процесу по всій популяції, визначається та мережа, похибка якої найменша. Вона повертається з функції.

1. Модуль backpropagation\_functions.py

У цьому модулі втілено функції, що використовуються для навчання зворотним поширенням помилки.

* set\_errors(neur\_arr, err) – поширює значення помилки err всіма нейронами списку neur\_arr від вихідного шару до всіх нейронів попередніх шарів за таким алгоритмом: значення помилки кожного нейрона з вихідного шару передаються на попередній шар, помножені на вагу зв’язку між ними. Кожен нейрон підсумовує всі отримані зворотні сигнали та за таким самим принципом передає на попередній рівень. Робота функції завершується тоді, коли пройдено всі шари мережі;
* update\_delta\_w(neur\_arr, eta, delta\_w\_arr, delta\_w\_bias\_arr, iteration, batch) – оновлює списки значень delta\_w\_arr, на які необхідно скоригувати ваги нейронів для навчання мережі, та delta\_w\_bias\_arr, delta\_w\_bias, на які необхідно скоригувати зміщення нейронів для навчання мережі. Переглядаючи список neur\_arr з вихідного шару, функція розглядаючи ітераційно всі його нейрони, переглядає всі їхні зв’язки з нейронами попереднього шару, обчислює значення функції активації цих нейронів від їх локального поля, множить їх на отриману похибку, вихід від нейрона попереднього шару з розглянутим нейроном вихідного шару та на коефіцієнт навчання нейронної мережі. Отримане значення додається до вже наявного для зв’язку між цими двома нейронами в списку значення яке, у випадку досягнення значення batch, додається до ваги цього зв’язку, таким чином коригуючи його. Процес продовжується для всіх шарів мережі, на вихід повертаються оновлені списки delta\_w\_arr та delta\_w\_bias\_arr.

1. Модуль learning\_algorithms.py

У цьому модулі знаходяться функції, що запускають обидва алгоритми навчання – зворотного поширення помилки та генетичний. Для них створено дві відповідні функції:

* backpropagation(neur\_arr, res, eta, err, batch, iteration, delta\_w\_arr, delt\_w\_bias\_arr) – для кожного нейрона зі списку neur\_arr обчислює значення , на яке змінюється вага, та сумує їх, щоб після досягнення значення batch додати та оновити вагу нейрона, додавши до неї відповідне значення . Для цього використовуються функції модуля backpropagation\_functions set\_errors та update\_delta\_w. Перша поширює значення помилки шарами нейронної мережі від останнього до першого. Друга отримує на вхід і повертає параметри delta\_w\_arr – список, що містить усі значення , на які потрібно оновити ваги зв’язків між нейронами та delta\_w\_bias\_arr – список, що містить усі значення , на які потрібно оновити зміщення нейронів, у якому зберігаються всі значення для нейронів у цій партії та оновлює їх. Після досягнення значення batch ці списки обнулюються. По завершенню роботи для кожного нейрона запускається функція класу to\_zero;
* genetic(neur\_arr, neur\_layer\_arr, set\_length, set\_data, set\_res, w\_bottom, w\_upper, func\_arr, func\_der\_arr, crossover\_prob, mutation\_prob, population\_length) – задається довжина популяції population\_length, за допомогою функції create\_initial\_population (модуль genetic\_algorithm\_functions) утворюється словник, який містить відповідну до значення параметру population\_length кількість особин популяції у вигляді ключів та їхні значення пристосованості у вигляді значень. Проводиться нормалізація значень цього словника за допомогою функції norm\_fitness (модуль genetic\_algorithm\_functions). Нові особини утворюються за допомогою функції offspring\_creation (модуль genetic\_algorithm\_functions.py), що використовує значення ймовірності схрещування crossover\_prob та ймовірності мутації mutation\_prob. Після заповнення нового покоління популяції кількістю особин population\_length, з них обирається найкраща за допомогою функції choose\_best\_network (модуль genetic\_algorithm\_functions) (модуль genetic\_algorithm\_functions). Ця мережа повертається з функції.

1. Модуль activation\_functions.py

Цей модуль зберігає декілька функцій активації та їхні похідні. Містить три додаткові параметри – alpha\_lin – коефіцієнт для лінійної функціїї, alpha\_LU – коефіцієнт для функцій ReLU, Leaky ReLU, ELU, SELU, lambda\_LU – коефіцієнт для функції SELU. Включені функції активації, де x – вхідне значення нейрона:

* Linear(x) – лінійна функція alpha\_lin\*x;
* Linear\_der(x) – похідна лінійної функції alpha\_lin;
* ReLU(x, \*args) – функція ReLU. Якщо отримано параметр args, то виконує роль її похідної;
* ReLU\_der(x) – похідна функції ReLU, викликає функцію ReLU, передаючи їй x як параметр входу та як параметр args;
* ReLU\_x6(x) – функція ReLU6;
* ReLU\_x6\_der(x) – похідна функції ReLU 6;
* Leaky(x) – функція Leaky ReLU;
* Leaky\_der(x) – похідна функції Leaky ReLU;
* ELU(x) – функція ELU;
* ELU\_der(x, \*args) – похідна функції ELU. Якщо отримано необов’язковий параметр args, то виконує роль похідної функції SELU;
* SELU(x) – функція SELU;
* SELU\_der – похідна функції SELU, викликає функцію ELU\_der, передаючи їй x як параметр входу та як параметр args;
* sigmoid(x) – логістична функція активаії;
* sigmoid\_der(x) – похідна логістичної функції активації;
* Tanh(x) – функція гіперболічний тангенс;
* Tanh\_der(x) – похідна функції гіперболічний тангенс, викликає функцію Tanh у вигляді формулі ;
* softplus(x) – функція SoftPlus;
* softplus\_der(x) – похідна функції SoftPlus, викликає функцію sigmoid.
  1. Інструкція користувача

Після запуску програма надає інформацію про кількість утворених рівнів нейронної мережі та кількість нейронів на кожному з них. Після цього розпочнеться навчання за допомогою алгоритму зворотного поширення помилки. По його завершенню на екран буде виведено середньоквадратичні похибки на навчальній і тестовій вибірках та побудовано три графіки: порівняння очікуваного й отриманого результату на навчальній вибірці, криву навчання мережі та порівняння очікуваного й отриманого результату.

Далі розпочнеться навчання за допомогою генетичного алгоритму. По його завершенню на екран буде виведено середньоквадратичні похибки на навчальній і тестовій вибірках та побудовано три графіки: порівняння очікуваного й отриманого результату на навчальній вибірці, криву навчання мережі та порівняння очікуваного й отриманого результату.

1. АНАЛІЗ РЕЗУЛЬТАТІВ РОЗВ’ЯЗАННЯ ЗАДАЧІ

Найкращі результати протягом тестування було отримано на мережі, з 2 проміжними шарами, що містять 2 та 1 нейрони, відповідно. Активаційні функції нейронів вхідного та вихідного шарів – ELU з коефіцієнтом , що дорівнює . Активаційна функція нейронів проміжних шарів – гіперболічний тангенс.

Після навчання за допомогою алгоритму зворотного поширення помилки, було отримано похибку мережі на навчальній вибірці 0.034. Порівняння результату роботи мережі з очікуваним результатом на навчальній вибірці продемонстровано на рисунку 4.1. Крива навчання, продемонстрована на рисунку на 4.2, на цій вибірці досить гладка, показує поступовий градієнтний спуск до оптимального значення. Похибка на тестовій вибірці дорівнює 0.026. Порівняння результату роботи мережі з очікуваним результатом на тестовій вибірці продемонстровано на рисунку 4.3.

Навчання за допомогою генетичного алгоритму привело до похибки на навчальній вибірці 0.038. Порівняння результату роботи мережі з очікуваним результатом на навчальній вибірці продемонстровано на рисунку 4.4. Крива навчання, показна на рисунку 4.5, вказує на певну хаотичність процесу, оскільки має велику кількість нерівностей і стрибків значень похибки, однак алгоритм має здатність призводити до більш-менш оптимального результату. На тестовій вибірці похибка становить 0.028. Порівняння результату роботи мережі з очікуваним результатом на тестовій вибірці продемонстровано на рисунку 4.6.

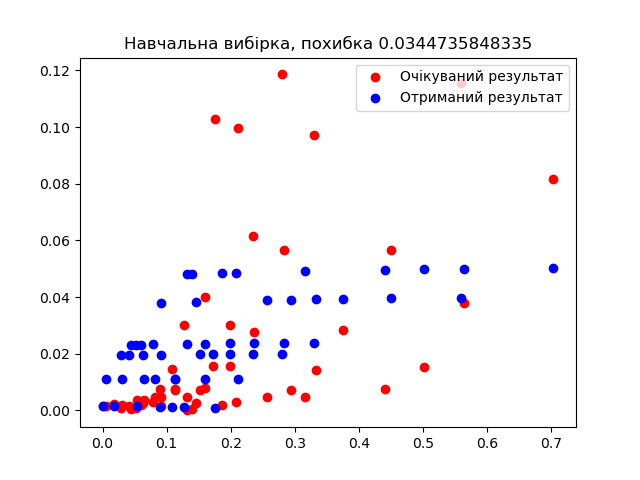


Рисунок 4.1 – Порівняння очікуваного та отриманого за допомогою навчання алгоритмом зворотного поширення помилки вигляду залежності (вісь y) від (вісь x) на навчальній вибірці

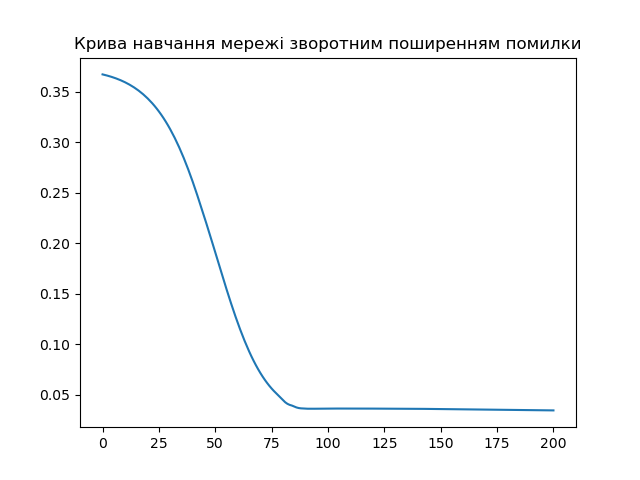
**

Рисунок 4.2 – Крива навчання нейронної мережі за допомогою алгоритму зворотного поширення помилки, вісь x – епоха навчання мережі, y – величина середньоквадратичної похибки

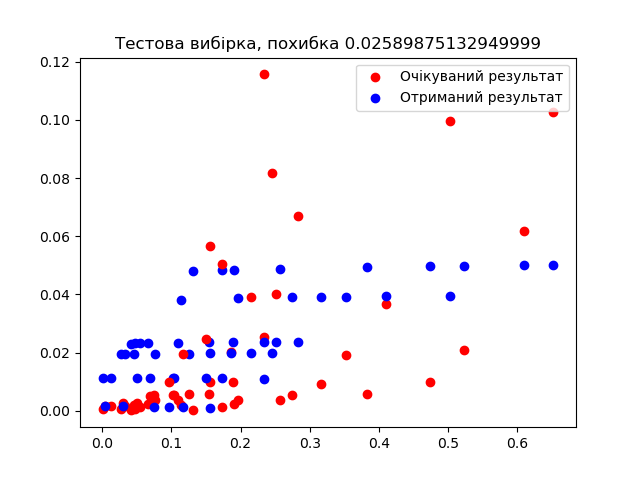


Рисунок 4.3 – Порівняння очікуваного та отриманого за допомогою навчання алгоритмом зворотного поширення помилки вигляду залежності (вісь y) від (вісь x) на тестовій вибірці

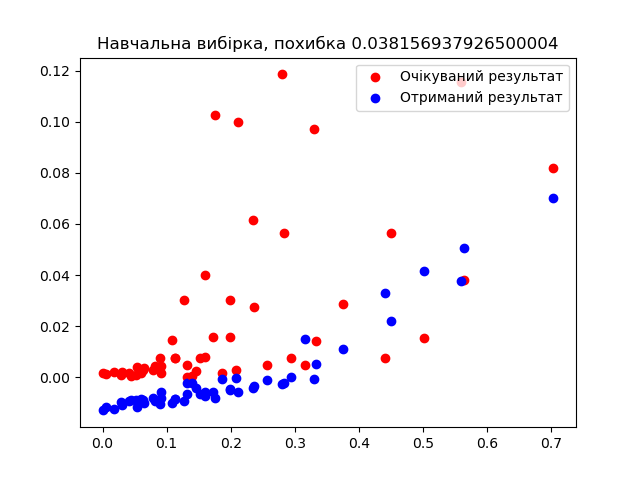


Рисунок 4.4 – Порівняння очікуваного та отриманого за допомогою навчання генетичним алгоритмом вигляду залежності (вісь y) від (вісь x) на навчальній вибірці

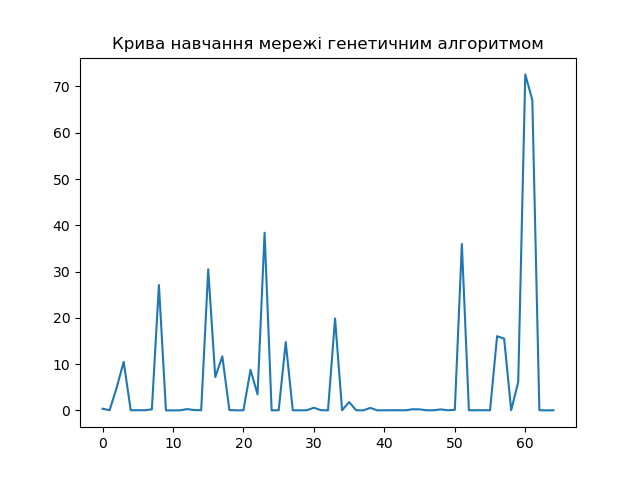


Рисунок 4.5 – Крива навчання нейронної мережі за допомогою генетичного алгоритму, вісь x – епоха навчання мережі, y – величина середньоквадратичної похибки

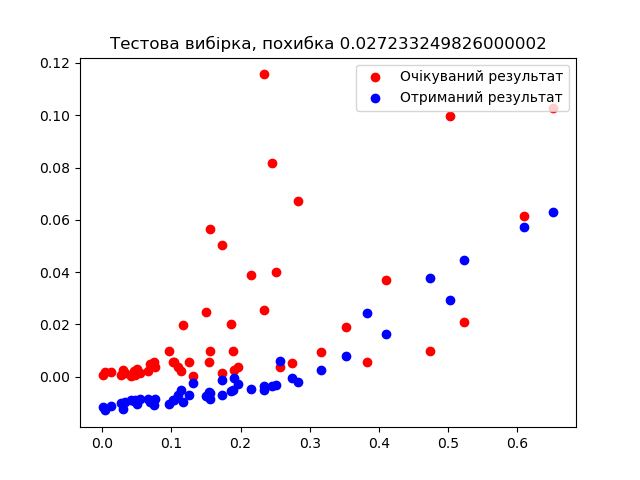


Рисунок 4.6 – Порівняння очікуваного та отриманого за допомогою навчання генетичним алгоритмом вигляду залежності (вісь y) від (вісь x) на тестовій вибірці

# ВИСНОВКИ

У ході даної роботи було сформульовано математичну постановку задачі про визначення можливості руйнування структурних елементів, розглянуто сфери її застосування для промисловості та задач із реального життя.

Розглянуто теоретичні принципи й методи машинного навчання, різні його види, як-от: навчання з учителем, навчання без учителя, навчання з підкріпленням, та різні моделі, що реалізують їх: дерева ухвалення рішень та їхні види (підсилені дерева, випадкові ліси, обертові ліси), метод опорних векторів, метод *k*-найближчих сусідів, нейронні мережі.

Розглянуто теоретичні принципи побудови та роботи нейронних мереж, їх математичні моделі, описано різні види топології та архітектури нейронних мереж, різні функції активації, їхні переваги та недоліки. Описано алгоритм навчання під назвою алгоритм зворотного поширення помилки, розглянуто проблему зникання градієнту в навчанні нейронних мереж.

Описано еволюційні алгоритми, їх теоретичні принципи та види використання в нейронних мережах. Зосереджено увагу на генетичних алгоритмах, їх застосуванні для навчання нейронних мереж.

Розглянуто принципи появи втомних тріщин, деякі моделі задачі про наростання втомних тріщин, перелічено вхідні параметри, що використовуються для них, відмінності між кожною з моделей.

Створено та протестовано програмний код, що утворює нейронну мережу для визначення залежності швидкості наростання вхідної тріщини від параметрів та , утілює процес її навчання за допомогою алгоритмів зворотного поширення помилки та генетичного, порівняно їх результати за допомогою графіків і значень похибки.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. M. I. Jordan, T. M. Mitchell. Machine learning: Trends, perspectives, and prospects // Science. 2015. Т.349, № 6245. С. 255-260. URL: <https://www.science.org/doi/10.1126/science.aaa8415>
2. Miroslav Kubat, Ivan Bratko, Ryszard Michalski. Machine Learning and Data Mining: Methods and Applications. Chichester: Wiley. 1998, 456 с.
3. А. В. Литвин. Розробка дерев рішень для прогнозування фінансової кризи в страхових компаніях України // Наукові записки. 2015. Т. 172. С. 59-64. URL: <http://ekmair.ukma.edu.ua/handle/123456789/6795>
4. Tom M. Mitchell. Machine Learning. McGraw-Hill Science/Engineering/Math. 1997. 421 с.
5. Yann Coadou. Boosted Decision Trees and Applications // EPJ Web of Conferences. 2013. Т. 55. 25 с. DOI: <https://doi.org/10.1051/epjconf/20135502004>
6. Byron P. Roe et al. Boosted Decision Trees as an Alternative to Artificial Neural Networks for Particle Identification // Nuclear Instruments and Methods in Physics Research. 2005. Т. 543, № 2-3. С. 139-142. DOI: [10.1016/j.nima.2004.12.018](https://arxiv.org/ct?url=https%3A%2F%2Fdx.doi.org%2F10.1016%2Fj.nima.2004.12.018&v=533ce3e2)
7. Byron P. Roe, Hai-Jun Yang, Ji Zhu. Boosted Trees, A Powerful Event Classifier. // [Statistical Problems in Particle Physics, Astrophysics and Cosmology](https://www.worldscientific.com/worldscibooks/10.1142/p446). 2006. 3 с. DOI: [10.1142/9781860948985\_0029](http://dx.doi.org/10.1142/9781860948985_0029)
8. Д. М. Вдовичинський, А. М. Родiонов. Відбір ознак за допомогою випадкового лісу у системі виявлення вторгнень. *XV Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених «Теоретичні i прикладні проблеми фізики, математики та інформатики»*, 25-27 травня 2017 року, м. Київ. Київ : ВПI ВПК «ПОЛIТЕХНIКА», 2017. С. 20-22. – Бібліогр.: 8 назв. DOI: <https://ela.kpi.ua/handle/123456789/20712>
9. Gregor Stiglic, Peter Kokolm Juan J. Rodriguez. Rotation of Random Forests for Genomic and Proteomic Classification Problems // Advances in Experimental Medicine and Biology. New York, NY: Springer, 2011. Т. 696. С. 211-221. DOI:  <https://doi.org/10.1007/978-1-4419-7046-6_21>
10. Степанюк, Є. Ю. Математичне та програмне забезпечення для аналізу потоків текстових даних: магістерська дисертація: 121 Інженерія програмного забезпечення / Степанюк Євгеній Юрійович. Київ, 2019. 109 с. DOI: https://ela.kpi.ua/handle/123456789/31665
11. Alex Graves. Supervised Sequence Labelling with Recurrent Neural Networks. // Studies in Computational Intelligence. Berlin, 2012. Т. 385. 146 с. DOI: https://doi.org/10.1007/978-3-642-24797-2
12. Emile Fiesler and Russell Beale. Handbook of Neural Computation. Taylor and Francis: New York, 1997. 436 с. DOI: <https://doi.org/10.1201/9780429142772>
13. S. Rajasekaran and G.A. Vijayalkshmi Pai. Neural Networks, Fuzzy Logic and Genetic Algorithms: Synthesis and Applications. PHI Learning Private Limited: New Delhi, 2003. 441 с.
14. Roshan Kaloni et al. Optimized Extreme Learning Machine // International Journal for Research in Applied Science & Engineering Technology, Т. 10, № 4, 2022, С. 1334-1341.
15. Tiago Matias et al. Genetically Optimized Extreme Learning Machine // *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013, 1-8 с. DOI: [10.1109/ETFA.2013.6647975](https://doi.org/10.1109/ETFA.2013.6647975)
16. D. Randall Wilson, Tony R. Martinez. The general inefficiency of batch training for gradient descent learning // Neural Networks, 2003. Т. 16, № 10, С. 1429-1451. DOI: <https://doi.org/10.1016/S0893-6080(03)00138-2>
17. Y.H. Zweiri, J.F. Whidborne, L.D. Seneviratne. A three-term backpropagation algorithm // Neurocomputing. 2003. Т. 50. С. 305-318. DOI: <https://doi.org/10.1016/S0925-2312(02)00569-6>
18. Ashay Singh and Ankur Singh Bist. Analysis of Activation Functions // Journal of Emerging Technologies and Innovative Research. 2019. Т. 6, № 2. C. 37-42. DOI: <http://www.jetir.org/papers/JETIR1902C08.pdf>
19. Ken Jon M. Tarnate, Dr. Madhavi Devaraj, Joel C. De Goma. Overcoming the Vanishing Gradient Problem of Recurrent Neural Networks in the Iso 9001 Quality Management Audit Reports Classification // International Journal of Scientific & Technology Research. 2020. Т. 9, № 3. С. 6683-6686. DOI: <http://www.ijstr.org/final-print/mar2020/Overcoming-The-Vanishing-Gradient-Problem-Of-Recurrent-Neural-Networks-In-The-Iso-9001-Quality-Management-Audit-Reports-Classification.pdf>
20. Pascanu, R., Mikolov, T., Bengio, Y. On the difficulty of training recurrent neural networks. // Proceedings of the 30th International Conference on Machine Learning in Proceedings of Machine Learning Research, 2013. Т. 28, № 3. С. 1310-1318. DOI: https://proceedings.mlr.press/v28/pascanu13.html.
21. Г. І. Великоіваненко, В. В. Корчинський, В. В. Чернишова. Дослідження ефекту перенавчання нейронних мереж на прикладі задачі аплікаційного скорингу // Нейро-нечіткі технології моделювання в економіці. 2016. №5. С. 3-23. DOI: [https://ir.kneu.edu.ua:443/handle/2010/20355](https://ir.kneu.edu.ua/handle/2010/20355)
22. A.P. Piotrowski, J.J. Napiorkowski. A comparison of methods to avoid overfitting in neural networks training in the case of catchment runoff modeling // Journal of Hydrology. 2013. Т. 476. С. 97-111. DOI: <https://doi.org/10.1016/j.jhydrol.2012.10.019>
23. Casper Hansen. Activation Functions Explained - GELU, SELU, ELU, ReLU and more. 2019. URL: <https://mlfromscratch.com/activation-functions-explained/> (дата звернення: 21.04.2022).
24. Tomasz Szandała. Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks // Bio-inspired Neurocomputing. Studies in Computational Intelligence, Т. 903. Singapore: Springer, 2020. 24 с. DOI: <https://doi.org/10.1007/978-981-15-5495-7_11>
25. Julia Garcia Gabello. Mathematical Neural Networks // Axioms. 2022. Т 11, №2: 80. DOI: <https://doi.org/10.3390/axioms11020080>
26. Zbigniew Michalewicz, Dipankar Dasgupta, Rodolphe G. Le Riche, and Marc Schoenauer. Evolutionary Algorithms for Constrained Engineering Problems. // Computers and Industrial Engineering. 1996. Т. 30, №4. С. 851-870. DOI: <https://doi.org/10.1016/0360-8352(96)00037-X>
27. Dan Simon. Evolutionary Optimization Algorithms. Hoboken, New Jersey: John Wiley & Sons, 2013. 784 с.
28. Joshua D. Knowles, David W. Corne. Chapter 13. Evolving Neural Networks for Cancer Radiotherapy // The Practical Handbook of Genetic Algorithms: Applications. New York, NY: Chapman and Hall/CRC, 2000. 544 c. DOI: <https://doi.org/10.1201/9781420035568>
29. Barbro Back, Teija Laitinen, Kaisa Sere. Neural Networks and Genetic Algorithms for Bankruptcy Predictions // Expert Systems with Applications. 1996. Т. 11, № 4. С. 407-413. DOI: <https://doi.org/10.1016/S0957-4174(96)00055-3>
30. Mijanur Rahman, Tania Akter Setu. An Implementation for Combining Neural Networks and Genetic Algorithms // International Journal of Computer Science and Technology. 2015. Т. 6, № 3. C. 218-222. DOI: <https://www.ijcst.com/vol63/2/48-tania-akter-setu.pdf>
31. Hans Albert Richard, Manuela Sander. Fatigue Crack Growth Detect – Assess – Avoid // Solid Mechanic and Its Applications. Springer Cham. 2016. Т.227. 292 с. DOI: https://doi.org/10.1007/978-3-319-32534-7
32. П.В. Ясній. Пластично деформовані матеріали: втома і тріщинотривкість, Львів: Світ, 1998, 292 с.
33. N. Pugno et al. A generalized Paris’ law for fatigue crack growth // Journal of the Mechanics and Physics of Solids. 2006. Т. 54, № 7. С. 1333-1349. DOI: <https://doi.org/10.1016/j.jmps.2006.01.007>
34. S. M. Beden, S. Abdullah, A. K. Arrifin. Review of Fatigue Crack Propagation Models for Metallic Components // European Journal of Scientific Research. 2009. Т. 28, № 3. С. 364-397.
35. Sanjin Kršćanski, Josip Brnić. Prediction of Fatigue Crack Growth in Metallic Specimens under Constant Amplitude Loading Using Virtual Crack Closure and Forman Model // Metals. 2020. Т. 10, № 7. 14 с. DOI: <https://doi.org/10.3390/met10070977>
36. Iryna Didych, Oleh Pastukh, Yuri Pyndus, Oleh Yasniy. Evaluation of Structural Elements Lifetime by Neural Network. // Acta Metallurgica Slovaca. 2018. Т. 24, № 1. С. 82-87. DOI:  <https://journals.scicell.org/index.php/AMS/article/view/250>.
37. J.R. Mohanty et al. Application of artificial neural network for predicting fatigue crack propagation life of aluminum alloys // Computational Materials Science and Surface Engineering. 2009. Т. 1, № 3. С. 133-138. DOI: <https://www.archicmsse.org/vol09_3/0931.pdf>
38. K. Zarrabi, W.W. Lu, A.K. Hellier. An Artificial Network Approach to Fatigue Crack Growth // Advanced Materials Research. 2011. №275. С. 3-6. DOI: <https://doi.org/10.4028/www.scientific.net/AMR.275.3>
39. Allavikutty Raja, Sai Teja Chukka, Rengaswany Jayaganthan. Prediction of Fatigue Crack Growth Behaviour in Ultrafine Grained AI 2014 Alloy Using Machine Learning // Metals. 2020. №10. 13 c. DOI: <https://doi.org/10.3390/met10101349>
40. Linxian Zhi. Neural Network-based Modelling and Prediction for Crack Growth Process in Alloys // Faculty of Science, Engineering and Technology Swinburne University of Technology Hawthorn, VIC 3122, Australia. 2019. 178 с. DOI: <https://researchbank.swinburne.edu.au/file/70a0e67a-a468-473c-8cdc-b6c320c91244/1/linxian_zhi_thesis.pdf>

Додаток А

Код програми мовою Python

* Модуль main:

import create\_network as cn

import get\_data as gd

import calculations as calc

import copy as c

print()

# неоновлювані протягом роботи параметри

error\_threshold = 0.01

epochs\_threshold = 200

weight\_bottom = -1

weight\_upper = 1

# для зворотного поширення помилки

eta = 0.005

# для генетичного алгоритму

crossover\_prob = 0.75

mutation\_prob = 0.25

population\_length = 100

ga\_weight\_bottom = -1

ga\_weight\_upper = 1

# оновлюються протягом роботи

neurons\_created = 0

neurons = []

set\_data, set\_res = gd.get\_dataset("data")

set\_data = gd.normalise(set\_data)

set\_res = gd.normalise(set\_res)

in\_parms = len(set\_data[0])

out\_parms = len(set\_res[0])

tr\_set\_data = set\_data[::2]

tr\_set\_res = set\_res[::2]

test\_set\_data = set\_data[1::2]

test\_set\_res = set\_res[1::2]

activ\_funcs, activ\_funcs\_ders = gd.get\_activation\_funcs()

layer\_number, neuron\_layer\_quantity = cn.get\_layer\_num(in\_parms, out\_parms)

neurons = cn.create\_neurons(in\_parms, out\_parms, layer\_number, neuron\_layer\_quantity, activ\_funcs, activ\_funcs\_ders, weight\_bottom, weight\_upper)

neurons\_1 = c.deepcopy(neurons)

# зворотне поширення помилки

u = 1

calc.main\_calculation(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, test\_set\_data, test\_set\_res, neurons, neuron\_layer\_quantity, eta, u, 0)

# генетичний

u = len(tr\_set\_data)

calc.main\_calculation(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, test\_set\_data, test\_set\_res, neurons\_1, neuron\_layer\_quantity, eta, u, 1, ga\_weight\_bottom, ga\_weight\_upper, activ\_funcs, activ\_funcs\_ders, crossover\_prob, mutation\_prob, population\_length)

* Модуль get\_data:

import numpy as np

import inspect as insp

import activation\_functions as af

def get\_dataset(file\_name):

f = open(f"{file\_name}.txt")

data = []

res = []

for row in f:

new\_row = row.strip().split(" ")

i = 0

while i < len(new\_row):

new\_row[i] = float(new\_row[i])

if i == 0:

temp\_arr = []

temp\_arr.append(new\_row[i])

elif i == 1:

temp\_arr.append(new\_row[i])

elif i == 2:

res.append([new\_row[i]])

i += 1

data.append(temp\_arr)

return data, res

def normalise(set\_list):

upper = 1

lower = 0

set\_list = np.matrix(set\_list).T

i = 0

while i < len(set\_list):

max\_value = np.amax(set\_list[i])

min\_value = np.amin(set\_list[i])

j = 0

while j < set\_list[i].size:

set\_list[i, j] = round(((set\_list[i, j] - min\_value)/(max\_value-min\_value)) \* (upper-lower) + lower, 6)

j += 1

i += 1

set\_list = set\_list.T.tolist()

return set\_list

def get\_activation\_funcs():

functions\_list = insp.getmembers(af, insp.isfunction)

activ\_funcs = []

activ\_funcs\_ders = []

i = 0

while i < len(functions\_list):

if i%2 == 0:

activ\_funcs.append(functions\_list[i][1])

else:

activ\_funcs\_ders.append(functions\_list[i][1])

i += 1

return activ\_funcs, activ\_funcs\_ders

* Модуль neuron\_class:

import random as r

class Neuron:

index = 0

layer = 0

bias = 0

error = 0

S = 0

activ\_func = 0

activ\_func\_der = 0

def \_\_init\_\_ (self, num\_ind, num\_layer, func, func\_der, \*args):

self.index = num\_ind

self.layer = num\_layer

if self.layer != 0 and len(args) > 0:

bias\_bottom = args[0]

bias\_upper = args[1]

self.bias = round(bias\_bottom + (r.random() \* (bias\_upper - (bias\_bottom))), 6)

self.weights = []

self.exit\_values = []

self.activ\_func = func

self.activ\_func\_der = func\_der

def \_\_repr\_\_(self):

return f"{self.index}\_{self.layer}"

def get\_index(self):

return self.index

def get\_layer(self):

return self.layer

def add\_random\_weights(self, next\_layer\_quant, \*args):

if len(args) > 0:

w\_bottom = args[0]

w\_upper = args[1]

else:

w\_bottom = -1

w\_upper = 1

i = 0

while i < next\_layer\_quant:

self.weights.append(round(w\_bottom + (r.random() \* (w\_upper - (w\_bottom))), 6))

i += 1

def add\_weights\_from\_arr(self, weight\_arr):

i = 0

while i < len(weight\_arr):

self.weights.append(weight\_arr[i])

i += 1

def get\_weight(self, ind):

return self.weights[ind]

def get\_weights(self):

return self.weights

def update\_weight(self, ind, value):

self.weights[ind] += value

def get\_bias(self):

return self.bias

def update\_bias(self, value):

self.bias += value

def set\_bias(self, value):

self.bias = value

def set\_error(self, value):

self.error += value

def get\_error(self):

return self.error

def get\_exit\_values(self):

return self.exit\_values

def set\_exit\_value(self, ind, value):

try:

self.exit\_values[ind] += value

except IndexError:

self.exit\_values.insert(ind, value)

def get\_exit\_value(self, ind):

return self.exit\_values[ind]

def update\_S(self, value):

self.S += value

def set\_S(self, value):

self.S = value

def get\_S(self):

return self.S

def set\_activation\_function(self, func):

self.activ\_func = func

def set\_activation\_function\_der(self, func):

self.activ\_func\_der = func

def get\_activation\_function(self):

return self.activ\_func

def get\_activation\_function\_der(self):

return self.activ\_func\_der

def get\_activation\_function\_value(self, x):

return self.activ\_func(x)

def get\_activation\_function\_der\_value(self, x):

return self.activ\_func\_der(x)

def to\_zero(self):

self.error = 0

self.S = 0

self.exit\_values = []

* Модуль create\_network:

import neuron\_class as nc

def get\_layer\_num(parms\_in, parms\_out):

neur\_layer\_arr = []

layer\_num = int(input("Введіть кількість прихованих рівнів нейронної мережі: "))

if layer\_num > 0:

i = 0

while i < layer\_num:

neur\_layer\_arr.append(int(input(f"Введіть кількість нейронів на прихованому рівні {i+1}: ")))

i += 1

print()

neur\_layer\_arr.insert(0, parms\_in)

neur\_layer\_arr.insert(len(neur\_layer\_arr), parms\_out)

print(f"Кількість рівнів: {layer\_num+2}")

print(f"Кількість нейронів на рівнях: {neur\_layer\_arr}")

print()

return layer\_num, neur\_layer\_arr

def create\_layer(num, quant, layer, func, func\_der, bias\_bottom, bias\_upper):

i = 0

arr = []

while i < num:

arr.append(nc.Neuron(quant+i, layer, func, func\_der, bias\_bottom, bias\_upper))

i += 1

return arr

def add\_weights(neur\_arr, neur\_layer\_arr, \*args):

i = 0

while i < len(neur\_arr)-1:

for j in neur\_arr[i]:

if len(args) == 2:

j.add\_random\_weights(neur\_layer\_arr[j.get\_layer()], args[0], args[1])

else:

j.add\_weights\_from\_arr(args[0][j.get\_index()-1])

print()

break

i += 1

def create\_neurons(parms\_in, parms\_out, layer\_num, neur\_layer\_arr, func\_arr, func\_der\_arr, \*args):

neurons\_created = 0

neur\_arr = []

# ELU: 0

# Leaky: 1

# Linear: 2

# ReLU: 3

# ReLU6: 4

# SELU: 5

# Sigmoid: 6

# Softplus: 7

# Tanh: 8

activ\_func\_index\_in = 0

activ\_func\_index\_hidden = 8

activ\_func\_index\_out = 0

i = 0

while i < layer\_num+2:

if i == 0:

neur\_arr.append(create\_layer(parms\_in, neurons\_created+1, i+1, func\_arr[activ\_func\_index\_in], func\_der\_arr[activ\_func\_index\_in], args[0], args[1]))

neurons\_created += parms\_in

elif i == layer\_num+1:

neur\_arr.append(create\_layer(parms\_out, neurons\_created+1, i+1, func\_arr[activ\_func\_index\_out], func\_der\_arr[activ\_func\_index\_out], args[0], args[1]))

neurons\_created += parms\_out

else:

neur\_arr.append(create\_layer(neur\_layer\_arr[i], neurons\_created+1, i+1, func\_arr[activ\_func\_index\_hidden], func\_der\_arr[activ\_func\_index\_hidden], args[0], args[1]))

neurons\_created += neur\_layer\_arr[i]

i += 1

if len(args) > 0:

if len(args) == 2:

add\_weights(neur\_arr, neur\_layer\_arr, args[0], args[1])

else:

add\_weights(neur\_arr, neur\_layer\_arr, args[0])

return neur\_arr

* Модуль learning\_algorithms:

import backpropagation\_functions as bf

import genetic\_algorithm\_functions as ga

def backpropagation(neur\_arr, res, eta, err, batch, iteration, delta\_w\_arr, delta\_w\_bias\_arr):

bf.set\_errors(neur\_arr, err)

delta\_w\_arr, delta\_w\_bias\_arr = bf.update\_delta\_w(neur\_arr, eta, delta\_w\_arr, delta\_w\_bias\_arr, iteration, batch)

for n in neur\_arr:

for m in n:

m.to\_zero()

return delta\_w\_arr, delta\_w\_bias\_arr

def genetic(neur\_arr, neur\_layer\_arr, set\_length, set\_data, set\_res, w\_bottom, w\_upper, func\_arr, func\_der\_arr, crossover\_prob, mutation\_prob, population\_length):

new\_population = []

population\_with\_err = ga.create\_initial\_population(population\_length, neur\_arr, neur\_layer\_arr, set\_length, set\_data, set\_res, w\_bottom, w\_upper, func\_arr, func\_der\_arr)

normalized\_fitness\_population = ga.norm\_fitness(population\_with\_err)

while len(new\_population) < population\_length:

chromosome\_1, chromosome\_2 = ga.offsprings\_creation(normalized\_fitness\_population, w\_bottom, w\_upper, crossover\_prob, mutation\_prob)

new\_population.append(chromosome\_1)

new\_population.append(chromosome\_2)

best\_neur\_arr = ga.choose\_best\_network(new\_population, neur\_arr, set\_data, set\_res)

return best\_neur\_arr

* Модуль calculations:

import learning\_algorithms as la

import matplotlib.pyplot as plt

def calculate\_layer(curr\_layer, next\_layer, in\_arr, learning\_algorithm):

res = []

i = 0

while i < len(curr\_layer):

j = 0

weight\_arr = curr\_layer[i].get\_weights()

while j < len(weight\_arr):

exit\_value = weight\_arr[j] \* in\_arr[i]

curr\_layer[i].set\_exit\_value(j, exit\_value)

curr\_layer[i].update\_S(exit\_value)

if i == 0:

res.insert(j, exit\_value)

else:

res[j] += exit\_value

j += 1

i += 1

j = 0

while j < len(res):

res[j] = next\_layer[j].get\_activation\_function\_value(res[j]+next\_layer[j].get\_bias())

j += 1

return res

def calculate\_result(neur\_arr, in\_arr, learning\_algorithm):

i = 0

while i < len(neur\_arr)-1:

in\_arr = calculate\_layer(neur\_arr[i], neur\_arr[i+1], in\_arr, learning\_algorithm)

i += 1

return in\_arr

def calculate\_error(exp\_res, res, learning\_algorithm):

i = 0

err = 0

err\_arr = []

while i < len(exp\_res):

diff = exp\_res[i]-res[i]

err += diff

if learning\_algorithm == 0:

err\_arr.append(diff)

i += 1

if learning\_algorithm == 0:

return err, err\_arr

else:

return err

def calculating\_cycle(set\_data, set\_res, neur\_arr, learning\_algorithm):

result = calculate\_result(neur\_arr, set\_data, learning\_algorithm)

if learning\_algorithm == 0:

error, error\_arr = calculate\_error(set\_res, result, learning\_algorithm)

return result, error, error\_arr

else:

error = calculate\_error(set\_res, result, learning\_algorithm)

return result, error

def learning\_cycle(neur\_arr, neur\_layer\_arr, set\_data, set\_res, eta, batch, learning\_algorithm, \*args):

result = []

i = 0

while i < len(set\_data):

err\_for\_backprop = []

if learning\_algorithm == 0:

res, error\_for\_i, err\_for\_backprop = calculating\_cycle(set\_data[i], set\_res[i], neur\_arr, learning\_algorithm)

else:

res, error\_for\_i = calculating\_cycle(set\_data[i], set\_res[i], neur\_arr, learning\_algorithm)

error\_for\_i = round(error\_for\_i, 6)

result.append(res)

if i == 0:

error = error\_for\_i\*\*2

else:

error += error\_for\_i\*\*2

if learning\_algorithm == 0:

delta\_w, delta\_w\_bias = la.backpropagation(neur\_arr, result, eta, err\_for\_backprop, batch, i, args[0], args[1])

for n in neur\_arr:

for m in n:

m.set\_S(0)

if (i+1)%batch == 0 and learning\_algorithm == 1:

neur\_arr = la.genetic(neur\_arr, neur\_layer\_arr, len(set\_res), set\_data, set\_res, args[0], args[1], args[2], args[3], args[4], args[5], args[6])

i += 1

error /= 2

if learning\_algorithm == 1:

return neur\_arr, result, error

return result, error, delta\_w, delta\_w\_bias

def learning\_process(error\_threshold, epochs\_threshold, set\_data, set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm, \*args):

error = 1

err\_arr = []

delta\_w = []

delta\_w\_bias = []

count = 0

if learning\_algorithm == 0:

for n in neur\_arr:

temp\_arr = []

for m in n:

temp\_arr.append(0)

delta\_w.append(temp\_arr)

delta\_w\_bias.append(temp\_arr)

while error > error\_threshold and count < epochs\_threshold:

if learning\_algorithm == 0:

result, error, delta\_w, delta\_w\_bias = learning\_cycle(neur\_arr, neur\_layer\_arr, set\_data, set\_res, eta, batch, learning\_algorithm, delta\_w, delta\_w\_bias)

else:

neur\_arr, result, error = learning\_cycle(neur\_arr, neur\_layer\_arr, set\_data, set\_res, eta, batch, learning\_algorithm, args[0], args[1], args[2], args[3], args[4], args[5], args[6])

err\_arr.append(error)

count += 1

return result, neur\_arr, err\_arr

def calculate\_set\_res(neur\_arr, set\_data, set\_res):

result = []

i = 0

while i < len(set\_data):

res, error\_for\_i = calculating\_cycle(set\_data[i], set\_res[i], neur\_arr, 1)

error\_for\_i = round(error\_for\_i, 6)

result.append(res)

if i == 0:

error = error\_for\_i\*\*2

else:

error += error\_for\_i\*\*2

i += 1

error /= 2

return result, error

def main\_calculation(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, test\_set\_data, test\_set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm, \*args):

in\_parm\_for\_graph = 1

if learning\_algorithm == 0:

print("Алгоритм зворотного поширення помилки")

elif learning\_algorithm == 1:

print("Генетичний алгоритм")

if learning\_algorithm == 0:

tr\_res, neur\_arr, tr\_err\_arr = learning\_process(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm)

else:

tr\_res, neur\_arr, tr\_err\_arr = learning\_process(error\_threshold, epochs\_threshold, tr\_set\_data, tr\_set\_res, neur\_arr, neur\_layer\_arr, eta, batch, learning\_algorithm, args[0], args[1], args[2], args[3], args[4], args[5], args[6])

tr\_err = tr\_err\_arr[-1]

tr\_res, tr\_err = calculate\_set\_res(neur\_arr, tr\_set\_data, tr\_set\_res)

tr\_err\_arr.append(tr\_err)

i = 0

while i < len(tr\_res):

j = 0

while j < len(tr\_res[i]):

tr\_res[i][j] = round(tr\_res[i][j], 6)

j += 1

i += 1

print()

print("Навчальна вибірка")

print(f"Похибка: {tr\_err}")

print()

plt.figure(1+learning\_algorithm\*3)

plt.title(f"Навчальна вибірка, похибка {tr\_err}")

i = 0

while i < len(tr\_res):

plt.scatter(tr\_set\_data[i][in\_parm\_for\_graph], tr\_set\_res[i], c = "red")

plt.scatter(tr\_set\_data[i][in\_parm\_for\_graph], tr\_res[i], c = "blue")

i += 1

plt.legend(["Очікуваний результат" , "Отриманий результат"], loc = "upper right")

plt.show()

plt.figure(2+learning\_algorithm\*3)

if learning\_algorithm == 0:

plt.title("Крива навчання мережі зворотним поширенням помилки")

elif learning\_algorithm == 1:

plt.title("Крива навчання мережі генетичним алгоритмом")

i = 0

x = []

while i < len(tr\_err\_arr):

x.append(i)

i += 1

plt.plot(x, tr\_err\_arr)

plt.show()

print("Тестова вибірка")

test\_res, test\_err = calculate\_set\_res(neur\_arr, test\_set\_data, test\_set\_res)

i = 0

while i < len(test\_res):

j = 0

while j < len(test\_res[i]):

test\_res[i][j] = round(test\_res[i][j], 6)

j += 1

i += 1

print(f"Похибка: {test\_err}")

print()

plt.figure(3+learning\_algorithm\*3)

plt.title(f"Тестова вибірка, похибка {test\_err}")

i = 0

while i < len(test\_res):

plt.scatter(test\_set\_data[i][in\_parm\_for\_graph], test\_set\_res[i], c = "red")

plt.scatter(test\_set\_data[i][in\_parm\_for\_graph], test\_res[i], c = "blue")

i += 1

plt.legend(["Очікуваний результат" , "Отриманий результат"], loc = "upper right")

plt.show()

* Модуль backpropagation\_functions:

def set\_errors(neur\_arr, err):

i = len(neur\_arr)-1

while i >= 0:

j = 0

while j < len(neur\_arr[i]):

if i == len(neur\_arr)-1:

neur\_arr[i][j].set\_error(err[j]\*neur\_arr[i][j].get\_activation\_function\_der\_value(neur\_arr[i][j].get\_S()))

else:

k = 0

while k < len(neur\_arr[i+1]):

neur\_arr[i][j].set\_error(neur\_arr[i][j].get\_weight(k)\*neur\_arr[i+1][k].get\_error())

k += 1

j += 1

i -= 1

def update\_delta\_w(neur\_arr, eta, delta\_w\_arr, delta\_w\_bias\_arr, iteration, batch):

i = len(neur\_arr)-1

while i > 0:

j = 0

while j < len(neur\_arr[i]):

k = 0

while k < len(neur\_arr[i-1]):

f\_der = neur\_arr[i-1][k].get\_activation\_function\_der\_value(neur\_arr[i-1][k].get\_S())

delta\_w\_temp = neur\_arr[i-1][k].get\_error()\*f\_der

delta\_w = eta\*delta\_w\_temp\*neur\_arr[i-1][k].get\_exit\_value(j)

delta\_w\_bias = eta\*delta\_w\_temp

try:

delta\_w\_arr[i-1][k] += delta\_w

except IndexError:

delta\_w\_arr[i-1].insert(k, delta\_w)

try:

delta\_w\_bias\_arr[i-1][k] += delta\_w\_bias

except IndexError:

delta\_w\_bias\_arr[i-1].insert(k, delta\_w\_bias)

if (iteration+1) % batch == 0:

neur\_arr[i-1][k].update\_weight(j, round(delta\_w\_arr[i-1][k], 6))

neur\_arr[i-1][k].update\_bias(round(delta\_w\_bias\_arr[i-1][k], 6))

delta\_w\_arr[i-1][k] = 0

delta\_w\_bias\_arr[i-1][k] = 0

k += 1

j += 1

i -= 1

return delta\_w\_arr, delta\_w\_bias\_arr

* Модуль genetic\_algorithms\_functions:

import create\_network as cn

import neuron\_class as nc

import calculations as c

import random as r

import numpy as np

import math as m

def make\_chromosome(neur\_arr):

chromosome = ()

for i in neur\_arr:

for j in i:

k = j.get\_weights()

for l in k:

chromosome += (l, )

chromosome += (j.get\_bias(), )

return chromosome

def create\_initial\_population(initial\_population\_length, neur\_arr, neur\_layer\_arr, set\_length, set\_data, set\_res, w\_bottom, w\_upper, func\_arr, func\_der\_arr):

population\_with\_err = {}

i = 0

while i < initial\_population\_length:

err = 0

if i == 0:

new\_neur\_arr = neur\_arr.copy()

else:

new\_neur\_arr = cn.create\_neurons(len(neur\_arr[0]), len(neur\_arr[-1]), len(neur\_arr)-2, neur\_layer\_arr, func\_arr, func\_der\_arr, w\_bottom, w\_upper)

j = 0

while j < set\_length:

res, err\_for\_j = c.calculating\_cycle(set\_data[j], set\_res[j], new\_neur\_arr, 1)

err += err\_for\_j\*\*2

j += 1

err /= len(set\_res)

chromosome = make\_chromosome(new\_neur\_arr)

population\_with\_err[chromosome] = err

i += 1

return population\_with\_err

def norm\_fitness(population):

fit\_arr = list(population.values())

chromosome\_arr = list(population.keys())

new\_dict = {}

exp\_sum = 0

i = 0

while i < len(fit\_arr):

exp\_sum += m.exp(fit\_arr[i])

i += 1

i = 0

while i < len(fit\_arr):

new\_dict[chromosome\_arr[i]] = m.exp(fit\_arr[i])/exp\_sum

i += 1

return new\_dict

def parent\_selection(normalized\_population):

# roulette wheel selection

parents = r.choices(list(normalized\_population.keys()), weights = list(normalized\_population.values()), k = 2)

return parents

def crossover(chromosome\_1, chromosome\_2):

len\_crossover = r.randint(1, round(len(chromosome\_1)/2))

quant\_crossover = r.randint(1, round(len(chromosome\_1)/(2\*len\_crossover)))

chromosome\_1 = np.array\_split(chromosome\_1, round(len(chromosome\_1)/(len\_crossover)))

chromosome\_2 = np.array\_split(chromosome\_2, round(len(chromosome\_2)/(len\_crossover)))

temp\_arr = []

i = 0

while i < len(chromosome\_1):

temp\_arr.append(i)

i += 1

ind\_arr = r.sample(temp\_arr, quant\_crossover)

for i in ind\_arr:

temp = chromosome\_1[i]

chromosome\_1[i] = chromosome\_2[i]

chromosome\_2[i] = temp

i = 1

while i < len(chromosome\_1):

chromosome\_1[0] = np.concatenate((chromosome\_1[0], chromosome\_1[i]))

chromosome\_2[0] = np.concatenate((chromosome\_2[0], chromosome\_2[i]))

del chromosome\_1[i]

del chromosome\_2[i]

chromosome\_1 = tuple(map(tuple, chromosome\_1))[0]

chromosome\_2 = tuple(map(tuple, chromosome\_2))[0]

return chromosome\_1, chromosome\_2

def mutation(chromosome, w\_bottom, w\_upper):

# random resetting

chromosome = list(chromosome)

quant\_mutation = r.randint(1, round(len(chromosome)/3))

temp\_arr = []

i = 0

while i < len(chromosome):

temp\_arr.append(i)

i += 1

ind\_arr = r.sample(temp\_arr, quant\_mutation)

for i in ind\_arr:

chromosome[i] = round(w\_bottom + (r.random() \* (w\_upper - (w\_bottom))), 6)

chromosome = tuple(chromosome)

return chromosome

def offsprings\_creation(normalized\_fitness\_population, w\_bottom, w\_upper, crossover\_prob, mutation\_prob):

chromosome\_1, chromosome\_2 = parent\_selection(normalized\_fitness\_population)

if 0 + r.random() \* (1-0) < crossover\_prob:

chromosome\_1, chromosome\_2 = crossover(chromosome\_1, chromosome\_2)

if 0 + r.random() \* (1-0) < mutation\_prob:

chromosome\_1 = mutation(chromosome\_1, w\_bottom, w\_upper)

if 0 + r.random() \* (1-0) < mutation\_prob:

chromosome\_2 = mutation(chromosome\_2, w\_bottom, w\_upper)

return chromosome\_1, chromosome\_2

def create\_new\_network(chromosome, neur\_arr):

i = 0

network = []

for i in neur\_arr:

layer = []

for j in i:

weights\_num = len(j.get\_weights())

neuron = nc.Neuron(j.get\_index(), j.get\_layer(), j.get\_activation\_function(), j.get\_activation\_function\_der())

neuron.add\_weights\_from\_arr(chromosome[0:weights\_num])

del chromosome[0:weights\_num]

if neuron.get\_layer != 0:

neuron.set\_bias(chromosome[0])

del chromosome[0]

layer.append(neuron)

network.append(layer)

return network

def choose\_best\_network(population, neur\_arr, set\_data, set\_res):

temp\_arr = population.copy()

population\_with\_err = {}

i = 0

while i < len(temp\_arr):

new\_neur\_arr = create\_new\_network(list(temp\_arr[i]), neur\_arr)

err = 0

j = 0

while j < len(set\_res):

res, err\_for\_j = c.calculating\_cycle(set\_data[j], set\_res[j], new\_neur\_arr, 1)

err += err\_for\_j\*\*2

j += 1

err /= len(set\_res)

population\_with\_err[population[i]] = err

i += 1

dict\_values\_list = list(population\_with\_err.values())

err = min(dict\_values\_list)

ind\_err = dict\_values\_list.index(err)

new\_neur\_arr = create\_new\_network(list(population[ind\_err]), neur\_arr)

return new\_neur\_arr

* Модуль activation\_functions:

import math as m

alpha\_lin = 2

alpha\_ReLU = 0.1

alpha\_ELU = 0.2

alpha\_SELU = 1.2

lambda\_SELU = 0.1

def Linear(x):

return alpha\_lin\*x

def Linear\_der(x):

return alpha\_lin

def ReLU(x, \*args):

if x < 0:

return 0

else:

if len(args) > 0:

return 1

else:

return x

def ReLU\_der(x):

return ReLU(x, x)

def ReLU\_x6(x):

return min(max(0,x), 6)

def Relu\_x6\_der(x):

if x < 0 or x > 6:

return 0

return 1

def Leaky(x):

if x < 0:

return alpha\_ReLU\*x

else:

return x

def Leaky\_der(x):

if x < 0:

return alpha\_ReLU

else:

return 1

def ELU(x):

if x < 0:

return alpha\_ELU\*(m.exp(x)-1)

else:

return x

def ELU\_der(x, \*args):

if len(args) == 0:

a = 1

else:

a = lambda\_SELU

if x < 0:

return a\*alpha\_ReLU\*m.exp(x)

else:

return a

def SELU(x):

if x < 0:

return lambda\_SELU \* (alpha\_SELU\*m.exp(x) - alpha\_SELU)

else:

return lambda\_SELU \* x

def SELU\_der(x):

return ELU\_der(x, x)

def Sigmoid(x):

return 1/(1+m.exp(-x))

def Sigmoid\_der(x):

return Sigmoid(x)\*(1-Sigmoid(x))

def Tanh(x):

return (m.exp(x) - m.exp(-x))/(m.exp(x) + m.exp(-x))

def Tanh\_der(x):

return 1 - (Tanh(x)\*\*2)

def Softplus(x):

return m.log(1+m.exp(x))

def Softplus\_der(x):

return Sigmoid(x)

Нормоконтроль пройдено

09.06.2022 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Наталія КОЗАКОВА